FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

MATT - Media Asset Tracking Tool

Tomás Fernandes Brandão Tavares

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Supervisor: Professora Maria Teresa Andrade

External Supervisor: Eng. Ricardo Serra

July 20, 2017

Abstract

In an era in which video and audio content is produced and consumed faster than ever, large amounts of data are generated every hour, day after day, year after year. This data needs to be stored and organized, in particular for the professional sector. Optimizing these processes may result in maximizing the profits of companies, which face a fierce market. This trend boosts the development of complex systems that allow handling and managing media assets easily. These tools are called digital asset management tools, and play an important role in television production workflows.

At the same time, the current development technologies are constantly evolving, originating new approaches to build software. These new technologies allied to efficient architectural patterns may result in powerful tools and solutions.

MATT (Media Asset Tracking Tool) is born from the combination of the two factors previously mentioned. On the one hand, a strong need for tools that support and help handling the huge amounts of data. And on the other hand, a growing number of possibilities to develop more efficient solutions. This project's architecture is based on microservices, developed with different technologies according to its purpose, which communicate through REST APIs. The tool exposes a web user interface that controls and monitors the system. This solution's architecture is deployed in a cloud environment.

This dissertation was proposed by MediaGaps, a company specialized in developing software for the television and entertainment industry. The main goal of this thesis is to develop a functional prototype of the described tool - MATT - with all the stages required to do so, such as research, design, development, testing and documenting.

This document will start with the results of the research done regarding the technologies, architecture design and methodologies used on the project associated to this dissertation. After that, all the tool and its modules will be described in detail, and in the end, the results, conclusions and future work will be presented.

Agradecimentos

A realização deste trabalho não seria possível sem o contributo de algumas pessoas às quais não poderia deixar de agradecer. Os meus agradecimentos:

A toda a equipa da MediaGaps. Ao Ricardo, à Joana, ao Paulo, ao José e ao Rui, por desde o primeiro segundo me terem feito sentir em casa. Por todo o conhecimento que partilharam comigo e por me ajudarem a dar o último passo nesta minha caminhada.

À Professora Maria Teresa Andrade, pelo desafio de embarcar nesta aventura e por toda a orientação que me deu para que a concluísse com sucesso.

Aos meus amigos, por nunca me deixarem esquecer o quão importante são os momentos de diversão e felicidade.

À Catarina, por partilhar comigo não só as pequenas vitórias, mas também os momentos mais difíceis. Por todo o apoio, carinho e paciência.

Em especial, à minha mãe, ao meu pai e à minha irmã, por serem os meus maiores exemplos e pelo enorme apoio e amor incondicional que sempre senti, não só nesta, mas em todas as etapas da minha vida.

A todos, um sincero obrigado.

Tomás Fernandes Brandão Tavares

Contents

List of Figures

List of Tables

Abbreviations and Symbols

API	Application Programming Interface
ART	Average Response Time
AWS	Amazon Web Services
CORS	Cross-origin resource sharing
CSS	Cascading Style Sheets
EBU	European Broadcasting Union
HTML	HyperText Markup Language
IaaS	Infrastructure as a Service
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
JWT	JSON Web Token
KLV	Key Length Value
MAM	Media Asset Management
MATT	Media Asset Tracking Tool
MXF	Media eXchange Format
PaaS	Platform as a Service
QoS	Quality of Service
SaaS	Software as a Service
SOA	Service Oriented Architecture
SMPTE	Society of Motion Picture and Television Engineers
TCP	Transmission Control Protocol
TV	Television
UI	User Interface
UMID	Unique Material Identifier
W3C	World Wide Web Consortium

Chapter 1

Introduction

The technological evolution felt in the last decades has completely revolutionized the information technologies panorama and, as a consequence, the daily routine of the vast majority of people. Data is produced, distributed and consumed faster than ever before, and it can be accessed anywhere at any time.

On the one hand, the proliferation of the internet was responsible for this trend, because it supported new communication channels. On the other hand, the appearance of new gadgets, such as smart phones and tablets, very easily acquirable by the consumer, also had an important role in this revolution, once they made it possible for everyone to produce and spread data.

Naturally, there was also a revolution in the way television is produced and consumed. To start with, the analog signal stopped being used with the appearance of the digital signal. This had immediate consequences in the way video and audio was recorded. Data that was once stored in magnetic tapes, started being organized as digital files. At the same time, the distribution networks experienced major improvements achieving higher transmission speed and larger bandwidth. There was also a great development in the capture, distribution and home equipment. In addition, new codification theories and achievements, made the appearance of new formats with higher definition and quality. Finally, the merging between the broadband and the traditional broadcast world, introduced new forms of interaction between the TV and the viewer, such as for example, through social networks.

This new paradigm changed the daily routine of the vast majority of the television consumers, who demand more efficient and complex systems, which should be able to keep up with the technological development and to offer a better experience to the user. These demands imposed by the users push television providers to search for innovative solutions to integrate on their systems so that its efficiency can be improved and they can deliver a better service to its costumers.

1.1 Context

This new paradigm comes with complex technological challenges attached. It is now necessary to find new ways to manage these complex systems and handle all the data they generate.

Taking as an example a real television channel or a production company, it is not difficult to imagine the huge amount of hours of video and audio, captured by different cameras and microphones every single day, day after day, year after year. In some cases, teams are numerous and spread along a wide geographical area, and still have to share and access the same content. In television, companies are constantly under competition, and being the first ones to report a certain event can be completely decisive in the audience of a channel, and consequently in the business results. Therefore, it is extremely important that this data is easily and quickly accessible. Besides that, it is equally important to store all this data, so that it can be used in the future. To do so, companies have to make considerable investments in memory capable of storing all these contents.

It can be easily concluded that handling such an amount of assets is no easy task, and that it would be even more complex without the support given by tools made exclusively with that purpose - digital asset management tools. TV producers are investing in this kind of tools so that they can manage their contents in a more efficient way, having their systems better organized, saving time and money.

Fortunately, the software development technologies have also experienced a great evolution over the last years. These technologies allied to efficient development methodologies and clever architectural patterns can enable the creation of powerful tools capable of supporting the TV production industry in handling their media assets.

In this context, the current dissertation was proposed by *MediaGaps*, a company specialized in the development of software for the television and entertainment industry. This is a research and development project, which integrates the company's strategy, working as a prototype for a future innovative tool, built based on the most recent technologies and methodologies.

1.2 Motivation and Objectives

This dissertation is born from the context and business needs described in the previous section, and it consists in the development of a prototype for MATT (Media Asset Tracking Tool).

The main goal of MATT is to monitor a network of servers, collect information from it, process it and present it to the user. To do so, the tool will analyze the MXF (Media eXchange Format) files inside the different servers, and extract specific information from its metadata. This information will then be processed and presented to the user through a web application. This will allow the user to have access, in real time, not only to the location of the MXF files, but also to the characteristics of each file as well as to information about the whole system. Accordingly, this tool will be extremely useful in a production environment where huge amount of information are spread through a vast network of machines and shared among numerous teams.

Figure 1.1 represents an example of a typical professional pos-production workflow, with already the inclusion of the MATT tools. In this figure, there are several sources of content capturing and doing the ingest of assets to a complex network of several servers. Then, various editing or post-production stations access those servers in order to use the assets and finally to broadcast

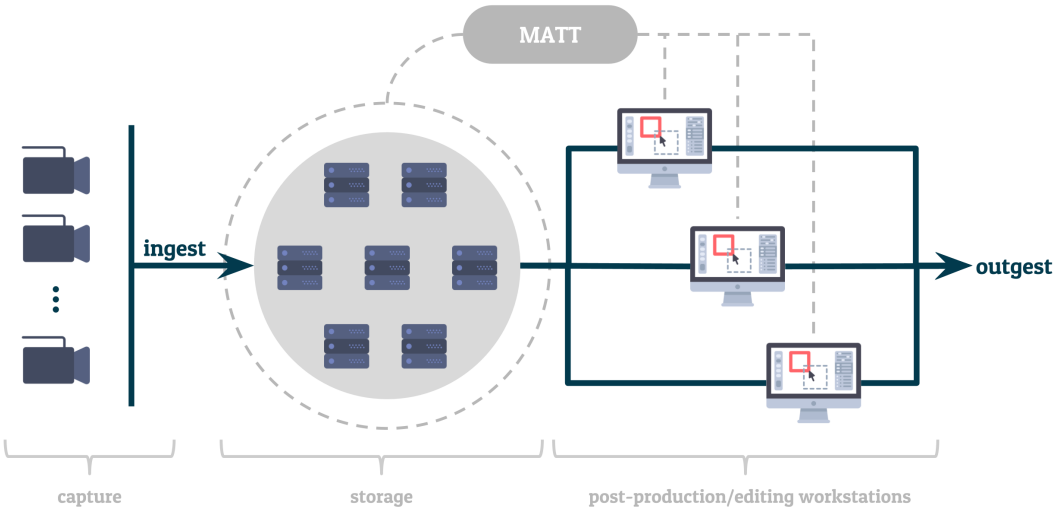

Figure 1.1: Workflow with MATT example

them or store them again. If there is no auxiliary tool, it is difficult to keep track of which assets are available on each server, detect eventual replicated files or have a statistical report in real time of what is happening on the network. This is where MATT can be extremely useful, because it will allow teams to easily understand what is the current state of the system, simplify the post production's operations, making them faster and minimizing error occurrence.

MATT development covers the complete software stack: from the data layer, where all the information is kept; the middleware layer, where the information is processed; and the user interface layer, where information is presented to the user. The architecture will be based in microservices, written in different programming languages and using different technologies, depending on the purpose of each microservice. As previously mentioned, this project integrates the company's strategy, and therefore, some modules will reuse software previously developed by MediaGaps, adapting it to this solution's requirements. This tool will also be *cloud ready*.

One of the challenges faced within this dissertation is the fact that it is expected the development of a full stack architecture, in quite a limited time. To achieve this goal, it is necessary a deep work of research to understand which are be the best technical choices and define the architecture of the solution.

The work to be developed in this dissertation is expected to deliver a fully functional prototype of the MATT system, supporting the identified requirements both from an operational viewpoint as well as functional. In this way, it will be able to serve as a successful proof-of-concept and, equally important, as the basis to be used by the company for a corresponding commercially exploitable application, eventually augmented in terms of functionality and performance.

1.3 Contributions

The challenge that is being addressed by this dissertation is not new and in fact can be considered as a generalized problem, currently experienced in the professional audiovisual content post-production sector. For this reason, a number of solutions have already been developed and experimented. As it will be explained in section 2.2.3, such kind of tools have also been specifically developed for the television industry. So one may question in what way does the solution developed in the scope of this dissertation differ from the ones already in the market, and how can it contribute to the improvement of the current state of the TV production industry.

Even though MATT shares some features with the MAM (Media Asset Management) tools, having some points in common with them, it has a different purpose. It differs from typical MAM tools not only due to the way it works, and how it was thought and built, but also because it is more focused on analyzing the system as a whole, rather than simply managing assets.

MATT combines a set of different technologies in a clever modular architecture, based on microservices. This architecture pattern allows not only a big scalability, re-usability and portability, but also the installation of the system in different environments. The majority of the systems used in the TV production industry are proprietary tools. Contrary to that, MATT is agnostic to the system used in the workflow and can be deployed in multiple heterogeneous platforms, therefore it is platform independent. This industry is also generally based on hardware and on-premises solutions. However, there is a recent trend to start adopting cloud technologies. MATT goes along with this trend, and may actually contribute to the virtualization of the industry.

In the prototype stage, it is expected that the user interface is a web application, which means that the UI can be accessed anywhere, anytime and in any device. However, the chosen architecture also allows the integration with different platforms, as for instance mobile solutions, in the future. Besides that, it also allows the integration with external APIs and products. The chosen technologies should guarantee an extremely performant tool, combining consolidated low level technologies, with recent and actual ones.

In the future, MATT may work with big data, machine learning and analytics techniques, in order to gather relevant information about the system, allowing a more intelligent management of it. At the end of the road, it may contribute to the automation of TV production systems.

This way, it is expected that this dissertation works as basis for a powerful tool which will, without any doubts,be disruptive and contribute to the improvement of the TV production workflow.

1.4 Document structure

This document will reflect and report the work developed during the internship at the company.

The first step of this work was to analyze the problem, understand what the different concepts were, and which technologies might be useful for the solution. This was achieved through a

research work that helped in the contextualization of the problem. The results of this work will be presented in chapter 2.

After presenting that information, the architecture of the solution will be described in detail and and each module explained and justified in chapter 3.

In the fourth chapter, the implementation details for each module will be presented, as well as the technological challenges faced during the development stage. Besides that, all the performance and load tests will be described.

Finally, in the last chapter the conclusions and results will be presented. It will also be described how this tool can be integrated in other systems and the future work that can be done using MATT as base for development.

Chapter 2

Review of the State of the Art

This chapter presents the results of a research about the different topics, technologies and methodologies that this dissertation comprises, describing the actual state of each of them.

2.1 Introduction

The television panorama changed through the last years, and the users have now new ways to consume it. However, this evolution was also felt in the different development technologies that support not only this industry, but many others. Additionally, new work methodologies and system architecture patterns are influencing the way software engineers develop their solutions.

To globally understand the problem addressed in this thesis, it was necessary to do a research which would not only describe and contextualize the TV production industry. but also analyze the current technologies and tools available to create the solution.

The results of this research can be found in this section and are organized according to the two fields previously mentioned. It starts with a section about the television industry and its changes over the last years, followed by a description of the different technologies that this dissertation comprises.

2.2 Professional Television Production

One can say that the value chain for television content production has been generally kept unchanged, for several years. However, in the current days, data is generated and consumed faster than ever before, reaching the consumers through several platforms. To keep up with this evolution, the television industry had to find new ways to adapt and improve in order to fulfill the high demands of consumers.

This evolution made the appearance of new ways of television consumption and new business models around this industry possible. Television is no longer a simple object owned by millions of families all around the world. It became possible to watch television in several screens, at anyplace, and anytime. The user has now control over what he wants to see at each instant, with

solutions on demand (non linear TV), and new concepts such as immersiveness and interactivity became more and more popular.

Despite all this evolution in the industry, it is possible to define a value chain for television production based in 3 main stages [9]:

- Content production;
- Aggregation;
- Delivery.

The contents are produced in the first stage, which includes steps such as filming, casting or scripting. After being produced, contents are sent to aggregation, which is where they are programmed and organized with other contents. Finally, they are distributed through the different platforms available until they reach the consumers.

2.2.1 TV Production Workflows

The TV production itself may be divided into three sub stages. The pre-production stage consists in the idealization and conceptualization of the content to be produced. The production stage is where the video and audio are actually captured. Finally, the post production stage comprises all the steps to process and edit the contents previously acquired.

In the past, the most common technology used in the production of TV contents were the magnetic tapes [10]. This systems were known as tape-based, because magnetic tapes were used to store the analogue video and audio signals. All the following processes were painful, slow and took human intervention. The processes of cataloging, organizing and storing had to be done manually, as well as the transition between the different stages of the workflow, because there was a physical resource that had to be shared.

In the post production stage, the editing was also a difficult step, due to the fact that the tape has a sequential and linear behaviour, hence the mechanisms to go through and mix different tapes were considerably more complex than in the systems used nowadays.

Additionally, once these were analogue and physical resources, the quality of the contents tended to degrade over the time, even when there were concerns regarding the archive conditions. Another common problem were the compatibility errors between different formats and equipment used to process the contents.

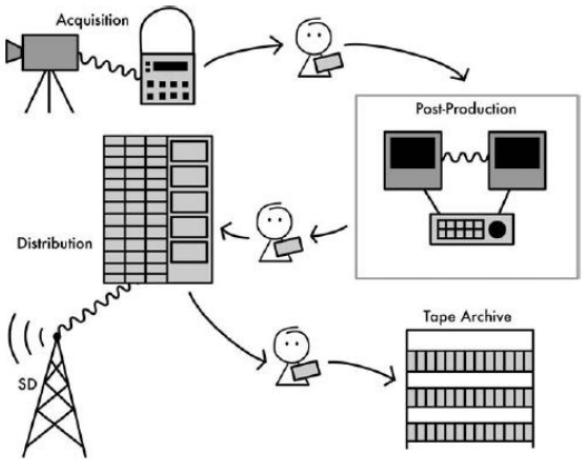

Figure 2.1: *Tape based* system workflow [1]

With the digital era, new production systems started to appear, using digital files as an alternative to the magnetic tapes. Due to this fact, these systems are known as *file-based* systems.

Similarly to the *tape-based* systems, in the digital paradigm, the workflow has as the starting point, the capture and acquisition of the video and audio signal, through hardware such as cameras and microphones. Typically, there is no compression applied to the video and audio, hence they are kept in a raw format. After this capture stage, it is time for the *ingest*. In the *ingest* step, the media contents (video and audio) are wrapped together with metadata, which is relevant data about these contents. This data will be very useful in the following stages, once it contains information about synchronization, subtitles, description of the files, among others. All this process is known by *wrapping*. The main goal of this stage is to group independent files in a single one with relevant information about it.

There are innumerable combinations of parameters and configuration possibilities for the different files to be wrapped. All these possibilities may change according to the video and audio formats, the final usage for the media contents, the equipment and software used or even the director personal preferences. All these variables make it hard to define a single and well established format used at the output of the *ingest* step. Despite that, the most commonly used format in the *wrapping* stage is called MXF (Media eXchange Format), and it was created precisely to uniform all these different possibilities to wrap contents. This format allows different set ups and configuration preferences, yet it guarantees interoperability among all the stakeholders of the workflow.

Finally, the contents are sent to a post production stage, where they will be edited and processed before they are broadcasted or sent to archive. This time, digital formats are being used, instead of magnetic tapes, which means all these operations are done using software instead of manual methods, which makes this process considerably simpler.

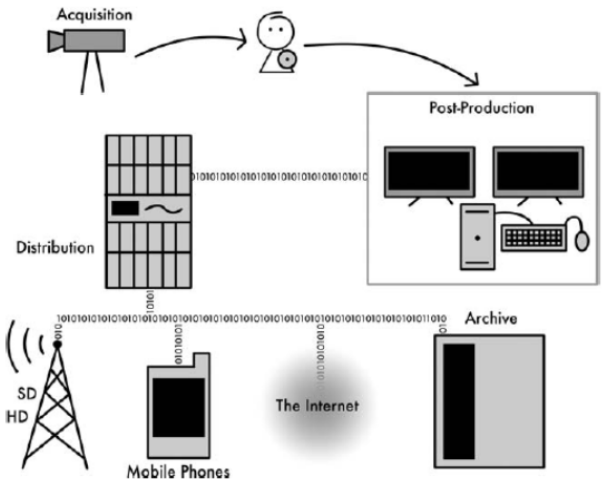

Figure 2.2: *File based* system workflow [1]

In this digital reality, the human intervention is substantially smaller than in *tape based* systems. The digital files are processed in powerful software, which make all the stages of cataloguing, editing, transporting or storing way easier, or even automatic.

2.2.2 MXF - Material eXchange Format

The MXF format appeared in the 90's, when the television scenario started approaching the new information technologies [10]. The magnetic tapes had stopped being used, and to replace them, the digital files, video servers and compression and editing software had appeared. With all these new technologies, there were different formats and terms popping up everyday, and this was creating interoperability problems between different manufacturers and producers. Due to this situation, it was made a joint effort between the *European Broadcasting Union (EBU)* and the *Society of Motion Picture and Television Engineers (SMPTE)*, to create an open and standard format. The first step to do so, was to identify the requirements for this format. After a few drafts, the first version of the Material eXchange Format (MXF) was released in 2004, under the name of *SMPTE S377M*

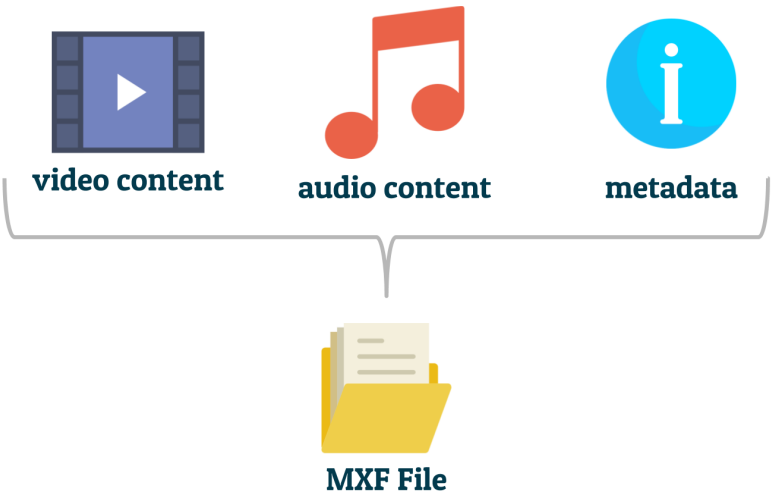

Figure 2.3: MXF Wrapping [2]

The MXF is a wrapper that groups media content with metadata [11]. This way, it does not act as an encoder. Instead, it transports media content in both raw or encoded formats, without changing the quality of the content. Therefore, one can say that it is format agnostic. MXF may be used in several applications, from storage to real time editing, allowing great flexibility. It can be adapted to specific applications without loosing efficiency.

The MXF packets are identified by a UMID (Unique Material Identifier) and can be one of two types:

- File packages - represent the origin files;
- Material packages - represent the desired output timeline.

Figure 2.4: MXF file structure [1]

As described in the figure 2.4, the MXF files are divided into at least two parts: header and footer. The header comprises the Structure Metadata, which transports important information such as the output timeline, technical characteristics, package ids and content descriptive information.

The footer indicates the end of the file. There can also exist a third part, located between the header and the footer, known as the File Body, which contains the file essence.

Taking a closer look, the MXF has a KLV (Key Length Value) structure. The first is a 16 bytes key, the length indicates the size of the element and the value is the actual data (video, audio or metadata).

The essence is transported inside the Essence Containers, that structure and organize the information, which can be an image, audio or metadata. These containers may be one of the following types:

- Frame-Wrapped - the frame and the corresponding audio are wrapped inside the same KLV;
- Clip-Wrapped - all the samples are wrapped inside the same KLV;
- Custom-Wrapped - the wrapping is customizable according to the application.

Finally, the Index Table is also part of the MXF file and does the mapping between a frame and an offset byte. This allows random access.

Due to all of these characteristics, the MXF has become the most used standard in professional television production environments, used by both hardware manufacturers and end-users.

2.2.3 Media Asset and Media Asset Management

To understand what Media Asset Management is, it is important to first know the definition of Media Asset. The definition of asset is strongly connected to a sense of property. In the digital world, this sense of property is usually associated with the intellectual property rights. Therefore, it is said that a digital file without the rights to use it is not an asset. [3] On the other hand, an asset is the combination of content and its usage rights [fig 2.5].

Figure 2.5: The definition of an asset [3]

After understanding what a media asset is, it seems now obvious to define Media Asset Management as the activity to manage, organize, catalogue or monetize a large library or repository of digital media assets. Even though it seems easy to define it, it is a rather complex process, being crucial in the workflow efficiency. This is why a lot of companies invest in powerful tools to support this process.

As previously mentioned, in the current times, it is easier than ever to produce new contents. Media assets can be of various types, such as pictures, videos, audio, motion graphics or text. Hence, one can conclude that the Media Asset Managements Tools usually deal with tremendous amounts of data. This fact makes the development of this kind of tools a real challenge. One of the main requirements of these systems is its performance. A big volume of files has to be processed and analyzed in the shortest possible period of time. This has a direct impact in the system's architecture pattern. Another important constraint is the usability of the system. The user must be able to find the asset he is looking for in a simple and easy way.

These kind of tools can have several practical applications, as for example a documentation and marketing material repository for regular companies, image specialized tools for photographers, or audio specialized tools for radios. Obviously there are MAM tools specially developed for the production of television content. Some examples are *Dalet Galaxy*, *Playbox MAM* or *VSN Explorer MAM*. These tools take an important role in the TV production workflows making them quicker, with less errors and supporting its stakeholders. This has a great impact not only in the efficiency of the workflow, but also from the business point of view.

2.3 Software Development Technologies

2.3.1 Service Oriented Architecture - *Microservices*

In the recent past, the majority of the applications were developed using a monolithic architecture. In this type of architecture it was normal to find long pieces of code responsible for a wide set of responsibilities, including functional and non functional requirements of the application [12]. This concept worked for low complexity applications, but as soon the complexity increased and new trends (namely the cloud) appeared, new approaches emerged.

One of the most popular approaches that appeared was the service oriented architecture, in particular the microservices. In this architecture, the application is divided into small modules with the goal of splitting a complex program into simpler tasks performed by individual microservices. This microservices should be simple, independent and should be able to communicate with each other through a language agnostic API [13]. This means that it does not matter in which programming language the microservices were developed, they should be able to communicate with each other. Besides that, the microservices should work independently from the server where they are hosted and the instance in which they start running. They should not expect an initial state of the application, working asynchronously.

This approach is a very efficient way to develop applications, because it makes it easy to modify, update or test it, allowing microservices to be reused among different applications, among companies, or even its commercialization. This also allows the creation of highly scalable and cloud based applications.

2.3.1.1 API Gateway

In a microservices architecture, one of the main questions to be raised is: "How does the client access each individual microservice?"[14] This is actually a very relevant question. In a complex system there may be hundreds of microservices, each of them with thousands of instances. These instances do not always run at the same location, and it would be very difficult for the user to, by his own, understand to which instance of each microservice should he communicate.

To answer this question, there must be a mechanism to orchestrate all this flow of requests and responses between clients and several microservices[4]. This orchestrator is known as the API Gateway. The API Gateway runs always in the same location, known by the user. All the requests go through the API Gateway which is responsible for routing the request to the correct service and retrieve back the response to the client. Besides this main responsibility, known as service discovery, the API Gateway may also be responsible for processes like authentication/security, protocol transformations, data aggregation, rate limiting or logging.

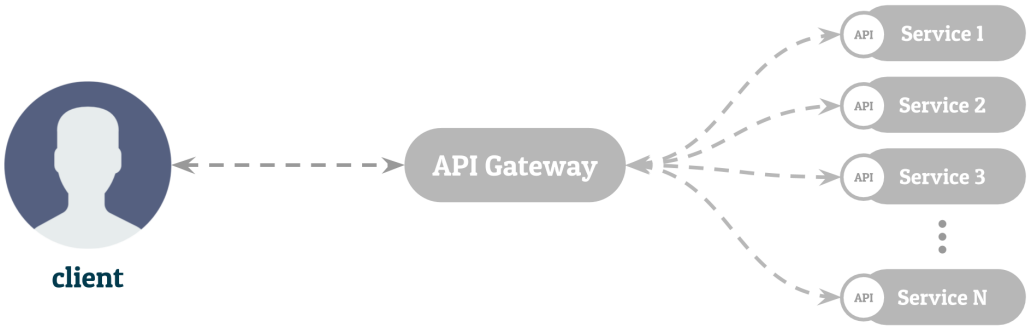

Figure 2.6: API Gateway behaviour [4]

To be able to dispatch all the requests the API gateway has to be able to know crucial information about each service. Firstly, it needs to know whether the service is up or down, namely if it is running or it is stopped. Then it needs to know where exactly is the service running [5].

Service Registry

The service registry is the place where information tracking all the active services and their location is stored, so that the requests can be successfully dispatched. It is usually a database, that is updated each time a service changes its current state.

There are two ways to do this: self-registration and third-party registration[5].

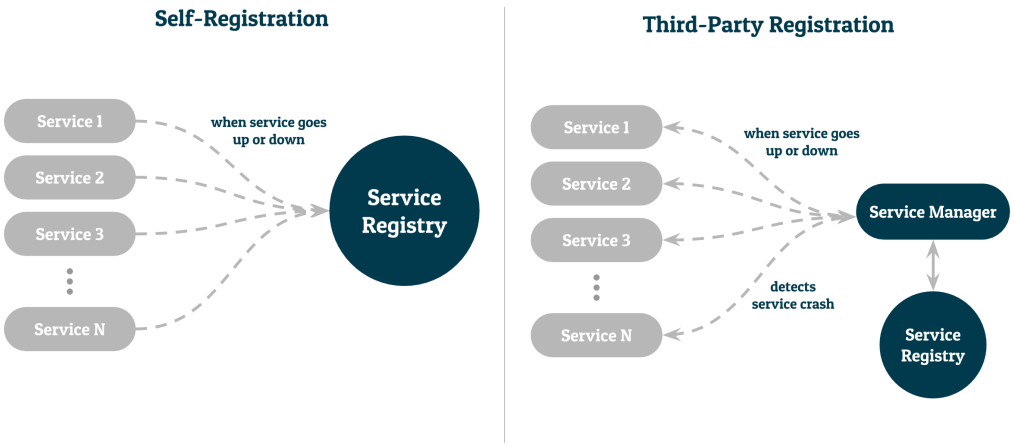

Figure 2.7: Service Registration [5]

In the self-registration mode, each microservice is responsible for registering itself directly with the service registry when it is going up. The opposite is expected when it is shutting down. All the information about the service should be provided by the service itself to the service registry

The other way to perform service registration is known as third-party registration and is done through a service that is responsible for managing all the other services, normally known as *service manager*. The service manager is responsible for registering a new service on the service registry, and check its state, looking for eventual failures. If this happens, the service manager may inform the service registry that the service is no longer available, or it may even be capable of restarting the service.

Service Discovery

Service Discovery is the process of localizing the desired service for a specific request made by the client. As previously mentioned, the information about the service location and status is kept at the service registry, hence at a certain point this registry has to be accessed. The service discovery may be one of two kinds: client-side discovery or server-side discovery[5].

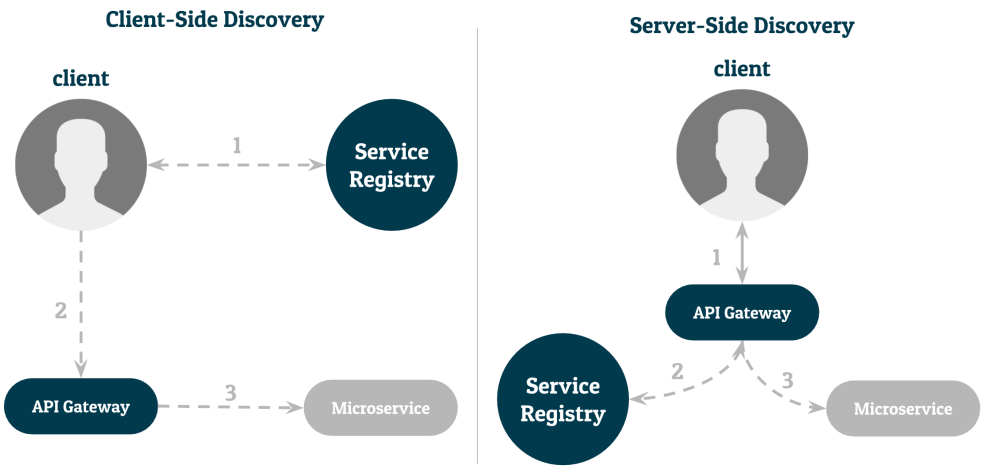

Figure 2.8: Service Discovery [5]

In the client-side discovery, the client has the responsibility to find out where the service is located. To do so, the client communicates directly with the service registry and only then does the actual request to the API gateway.

In the server-side discovery, the client makes a request to the API Gateway, which has the responsibility to connect to the service registry, discover where the service is running and dispatch the request.

Kong

Kong is an open source API gateway specialized in microservices management. It runs in front of a RESTful API and it is plugin based. It centralizes different features in the same module. Its architecture comprises the service registry which may be a Cassandra or a PostgreSQL database. It has a RESTful Administration API.

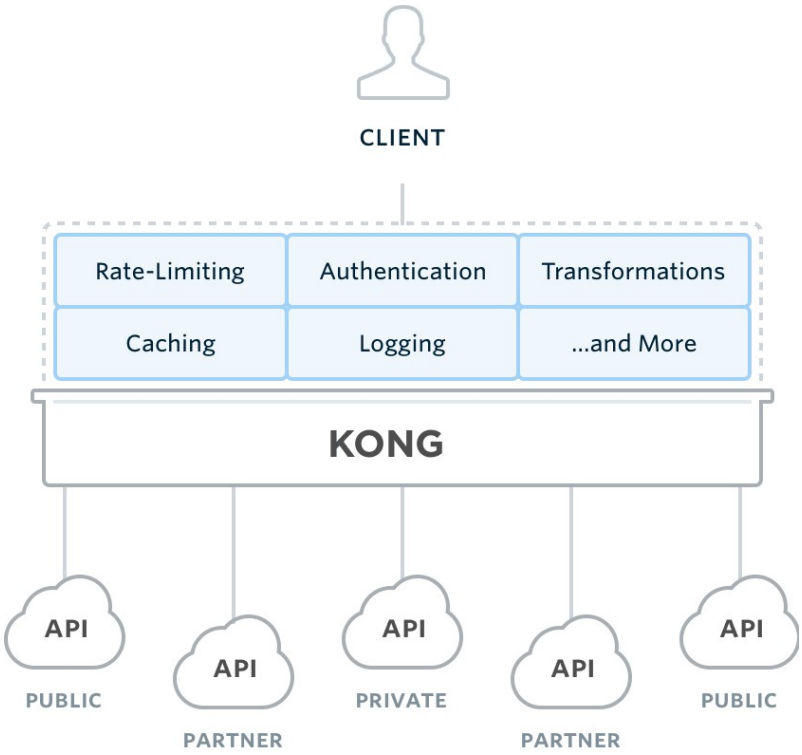

Figure 2.9: Kong architecture [6]

It is extremely performant and horizontally scalable, which means that for higher loads the performance can be kept simply by adding more machines. It is also modular. Plugins can be added or removed any time through the RESTful admin API. New plugins can be developed using the LUA programming language. It is platform agnostic, which means it runs on any infrastructure and it is extremely easy to set up.

When Kong is being used, all the requests made by the user to the API gateway go through Kong, and are only dispatched to the final service after executing all the active plugins.

2.3.2 Cloud Computing

In the recent past, cloud computing has become one of the most disruptive paradigms in the technological panorama. It is a model that allows access to powerful computing resources over the internet. These resources can be servers, storage, networks, applications or services. Typically, cloud computing providers, such as Amazon Web Services or Microsoft Azure, make available a platform through which users can easily configure and access their resources.

Cloud users, do not have to worry about maintenance, management or updating the resources they are using, and they are guaranteed an established QoS (quality of service). Costumers follow a pay-as-you-go model, which means they are billed according to the resources they have bought and its usage. This is an extremely flexible solution, which allows scalability. Users can buy a solution that fits their requirements, and increase the resources' capacities as their business grows[15].

Generally there are three service models in cloud computing[16]:

- Infrastructure as a Service (IaaS) - a model that provides hardware to the user (CPUs, memory, storage, networks);
- Platform as a Service (PaaS) - a model that provides a platform to get on-demand access to hardware resources;
- Software as a Service (SaaS) - a model that provides software applications (services) which users can subscribe.

Analyzing figure 2.10, it is possible to understand that the level of management required for the cloud resources varies depending on the model, but it is always smaller when compared to the traditional *on-premises* paradigm.

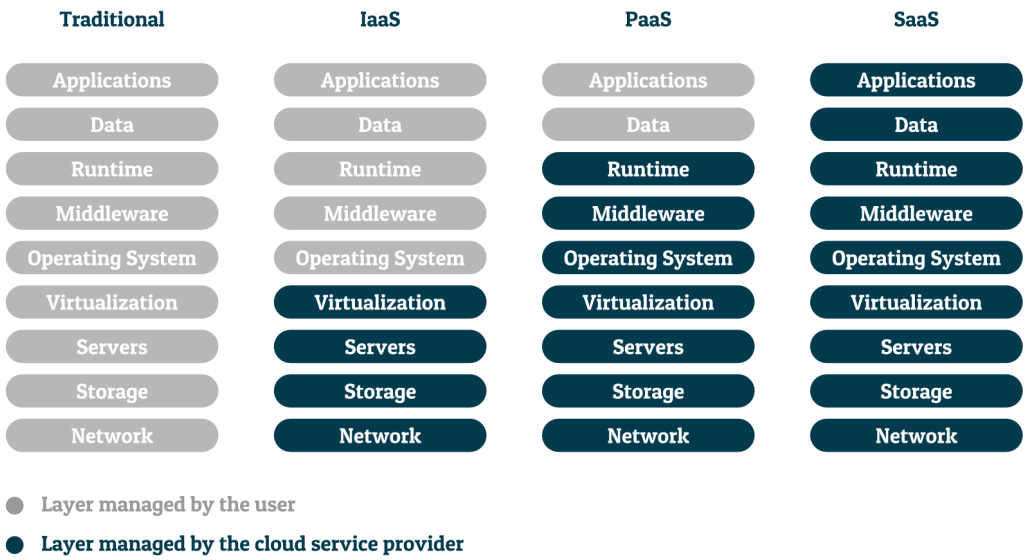

Figure 2.10: Traditional vs Cloud [7]

Some authors also make reference to another model named **Data as a Service (DaaS)**, which basically provides a database service.

There are 4 deployment models regarding the cloud privacy[16]:

- Private cloud - the cloud is deployed exclusively for a single organization;
- Community cloud - the cloud is deployed for exclusive use of a community of consumers, belonging to a known set of organizations;
- Public cloud - the cloud is deployed for for the general public and anyone is able to access it;
- Hybrid cloud - the cloud is composed by different cloud infrastructures with different privacy levels.

Generally, the cloud architecture brings several benefits for the user. It has reduced costs, it has an easy setup and it is extremely scalable. At the same time, there are some cons that should be taken in consideration. The user gets dependent on his service provider, looses control over his data which may bring security problems, gets dependent on an internet connection and on the availability of the service.

2.3.3 Relational and Non Relational Databases

The internet development and the increasing of its users and its applications' complexity, changed the database panorama over the last years. Nowadays, a database has to be able to store large amounts of data and have a good performance, otherwise it will have a negative impact on the application's efficiency. This is why it is so important to have a clever design.

In the past, the databases were typically relational. These databases are based in the Relational Model, and the information is kept in tables, that may be related with each other or not, and that are typically stored in the same server.

To perform the different operations, such as the insertion or removal of data in the database, commands written in SQL (Structured Query Languages) are used. Each query represents a transaction. The relational databases follow the ACID model which is described by the following properties[17]:

- Atomicity - a transaction either is successful and it is finished or it will fail;
- Consistency - the transactions change the database status to a new one, without loosing data consistency;
- Isolation - concurrent transactions can not affect each other, so the result should be the same as executing both transactions sequentially;
- Durability - after a transaction, its results should be kept and should not be lost by a system failure.

In relational databases, the way to scale a system is vertically, which means, it is done acquiring bigger and more powerful equipment. This approach is quite limited, not only because it is extremely expensive, but also because it will eventually reach a point in which it is not possible to scale the database. The response time from the database will be so big that the application will be impossible to use.

With the growing number of data stored online, boosted by the social networks' users increase, it was necessary to find alternatives for the relational databases, that would be able to handle such an amount of data, in a more efficient way. In 2009, there started to appear efforts to create a new paradigm in which data would be stored not only in one server, but spread over the internet instead. This was the origin for non relational databases. This kind of databases is no longer based in the ACID model. Instead, its principles follow the CAP theorem, created by E. Brewer.

This theorem says that, in a non relational database, only two of the following properties can be achieved:

- Consistency - all reading operations have access to the same information;
- Availability - a database request is always answered no matter the circumstances;
- Partitional tolerance - the system works independently of its partitions caused by physical network problems.

The BASE transactional model might be seen as a direct opponent to the ACID model and it is based on the CAP theorem:

- Basically Available - the data is available according to the CAP theorem. However, every request has an answer even if it is an error message;
- Soft Stage - meaning that the system state may change even if there is not a transaction. This may happen due to a background operation to guarantee consistency;
- Eventual Consistency - the consistency is not verified after each operation and request are always being answered.

In non relational databases, the information is no longer kept in tables. Instead, it may be kept in graphics, documents or associative arrays, which are spread over the internet [18]. A parallelism may be established to understand the relation between non relational databases and relational databases. If we consider a non relational database based on documents, we can say that a line of a relational database, may be seen as a document of a non relational database, and a table as a collection of documents.

These characteristics make non relational databases easily scalable an replicable, contrary to relational databases. The database access time is shorter as well when we compare this two realities. All these factors combined, result in a better global performance.

2.3.3.1 Non Relational Databases Comparison

Due to the previously mentioned characteristics, non relational databases have had a big increase in popularity and several options were released to the market. Even though these options are non relational databases, they can have numerous differences and may be used differently according to the application. The two solutions that at first sight seemed more adequate to this tool were MongoDB and Cassandra, and a research to both of them was made in order to understand which would better fit the solution.

MongoDB

Mongo DB was created in 2007, by a company named 10gen and it was presented as a database capable of scaling with the application's needs. [18] This service was not widely accepted by the developers' community, which forced the company to offer his service as open-source in 2009.

This system was based in a MySQL database, and it was changed into a distributed and document oriented database. To do so, it uses BSON (which is bynary JSON) to structure its documents.

It supports partial and total document updates, it has a rich and flexible search language, it is scalable, fast and easy to set up. There are several non official drivers that make it easy to integrate with other programming languages such as PHP, Java, C or Python.

All these reasons make it one of the most commonly used solutions.

Apache Cassandra

Apache Cassandra was released in 2008, by the Apache Software Foundation, as a open source and free non relational database.

Its data model is based in clusters or nodes, that are spread over different machines over the internet. Cassandra is capable of managing how the information is divided by its clusters, dealing with cluster registry and disconnection autonomously [19]. The clusters have keyspaces which contain column families. A column family is basically a set of wide rows which may contain up to 2 billion columns. The user is able to configure how many times a keyspace should be replicated inside a cluster and which strategy should it take [20].

Cassandra is known for being extremely performant even under a high number of requests. It is completely decentralized and scalable, and the queries are done through a specific query language called CQL (Cassandra Query Language). There are several drivers which facilitate the integration with the main programming languages.

2.3.4 C++

C++ is one of the most accepted and established languages in the developers community. Its beginning is deeply connected with the C language. It started when Bjarn Stoustrup tried to adapt C to an object oriented language, creating something named *C with classes*. [21]

In 1983, Stoustrup decided to adopt the actual name, and changed C with classes to C++, and in 1985 he published a work called *The C++ Programming Language*. The language had several iterations over the following years until the publication of its first international standard in 1998, the *ISO/IEC 14882:1998*. After that, several versions of C++ were released, with new libraries and modules.

Nowadays, it is widely used for low level and performant applications. It is a compiled language, which means it is converted directly to machine's native code, and therefore can achieve high values of speed and efficiency. It is unique because it can have such great performances and be object oriented. Similarly to the C language, it allows memory management, which is extremely useful for low level applications. Another important factor, is that it is extremely supported by external libraries.

It is expected a new revision for the ISO/IEC standard during the year of 2017, but no major changes are expected from this version, known as C++17. [22]

2.3.5 HTML e CSS

The HTML (HyperText Markup Language) is the markup language used for the webpages development. It is responsible for the structure of the content of a webpage. HTML history is deeply related to the world wide web history, even because both share the same creator, Tim Berners-Lee[23]. The first version of HTML was proposed in 1992 at the CERN and eventually it evolved with the contribution of several entities such as the IETF (Internet Engineering Task Force) or the W3C (World Wide Web Consortium)[24].

After requiring the server for a web page, the browser receives a HTML file as a response, and transforms it into visual information which is going to be displayed to the user.

The page structure is defined semantically, and the base elements are defined using tags described by these symbols "<>" [10]. There are several tags, which represent elements of a webpage. These elements may vary from interactive forms, texts, pictures among others. Besides that, the HTML is also responsible for the hyperlinks between different webpages.

The HTML5 is the last version of this standard and it introduced new elements, transitions and behaviours, allowing to make web applications more powerful[25].

Theoretically, HTML should only be used to structure a webpage content and not to define its visual characteristics. To do so, there is another language named CSS (Cascading Style Sheets). Both standards are defined by the W3C and should be used together.

Using CSS, to separate the content from the webpage design is a clever and efficient decision, because it allows to define several styles for multiple webpages, defining properties such as the color, fonts or layouts. To do so, it has two main concepts: selectors and properties. CSS uses selectors, to select which HTML element is it editing, and then it applies a set of properties to that element[26].

The first CSS version was released in 1996 and it has gradually evolved until it reached the current version number three. In the current days it is supported by all the browsers[27].

2.3.6 JavaScript

Together with HTML and CSS, JavaScript is one of the core languages for web development. It was created in 1995, developed by Brendan Eich at Netscape. In 1996 it was standardized by the ECMA International, based on the ECMA Script. Today it supported by every modern browsers, being a trademark of Oracle Corporation, and managed by the Mozilla Foundation [28].

It is a object oriented, prototype-based, dynamic, imperative and functional language. It can be used in the client side (browser), but also in the server side to perform, for example database queries [29].

Even though it is mostly used in web development, it can also have some other applications as for example video games development.

2.3.6.1 NodeJS

NodeJS is a Javascript runtime environment, developed in the Google V8 Javascript Engine and created in 2009 by Ryan Dahl. It is an asynchronous and event-driven programming language. This characteristic makes it one of the most used languages in the development of web servers, because it solves concurrence problems without using threads[30]. This means there are no dead lock problems, which makes it very scalable and capable of dealing with real time applications such as online gaming.

There is an open repository named npm (node package manager) in which hundreds of packets can be found and freely used. This tool is very useful because it allows developers to reuse code already done and contribute to the open source community.

2.3.6.2 ReactJS

ReactJS is a Javascript library, created in 2013 by Facebook, to develop user interfaces. It was born from the need to create user interfaces for high complexity applications, in which the data to present is variable over the time[31].

ReactJS is declarative which means that the views can be designed for each state, and will then be rendered automatically as soon as the system verifies that the data has changed.

It is also component based, which means, each component may be developed independently, being capable of generate its own state. It is also possible to render the views on the server side, combining ReactJS with NodeJS.

Components are typically written in JSX, which is a syntax based in Javascript, that allows the combination of HTML with Javascript.

2.4 Conclusion

The research previously presented is an important starting point for the development of the project associated to this dissertation and was extremely helpful to develop a critical approach to the design and the development of the solution.

To start with, it is extremely useful to understand what kind of solution should be developed, what was used before this kind of tools were available and why is it important to have this applications in the TV production workflows. It was helpful to clarify which other solutions are available in the market, and how can this tool make a difference when compared to the others. In particular, even though this is a prototype, this solution should be focused on being highly scalable and compatible with future work developed not only, but also by the company. It should be focused on the internet and cloud ready, so that it is accessible by different platforms, equipment and in different circumstances. It should be very performant and user friendly, so that it can be used in real production environments and increase the efficiency of the workflow.

After reflecting over the tool's contextualization, this research was crucial to start taking decisions over the architecture to be implemented. Since this tool should be modular and scalable, it

makes sense to resort to a microservices architecture. To develop these services, it was important to review the several languages available and understand which were more indicated for each module. Finally, this tool will handle large amounts of data, so it was mandatory to do some research on the different database systems.

In the end, all the information was gathered and the decisions to develop the solution could be taken in a informed a solid way. In the next chapter, all these decisions, reflected in the system architecture, will be described in detail.

Chapter 3

System Specification

In this chapter a detailed description of the solution will be made. The requirements and use cases will be taken as a starting point, and will lead to the design of the system architecture. Each module of the solution will be analyzed and described.

3.1 Introduction

Before starting writing code and developing the solution, it was important to do some reflection about this dissertation and understand why is this tool useful, who will use it and in which context or what features should it have.

This reflection was very important to start drafting possible solutions, architecture patterns and choosing technologies to use.

The majority of the questions previously presented are answered in chapter number 1 of this document. It is possible to understand the reasons that support the development of this kind of tool. In general, this tool should be able to make a report of a network of machines that contain a big volume of media assets, through a web application. This report is needed due to the huge amounts of assets and information in those machines, and will be extremely useful for an user, for instance an editor, looking for a specific asset among that network. One use case of MATT is explicitly described chapter 1, using figure 1.1 as a real example of a workflow that uses this tool.

Some other questions are not that general and demand a more detailed answer. It is important to predict how should this tool behave and how can its architecture be thought so that it is a long term solution. To do so, a list of requirements was drafted. This requirements worked as a lighthouse, to guide in the development of the solution. However, this was an iterative process which suffered, and still can suffer changes as the project grows and is developed.

3.1.1 Requirements

The requirements allow the understanding of which features and behaviours are expected from MATT, and at the same time, which non functional characteristics should be guaranteed.

3.1.1.1 Functional Requirements

The functional requirements are a set of features and actions that the system should be able to guarantee. After some reflection, these were the identified functional requirements, at this prototype stage:

Table 3.1: Functional Requirements

01	List all the machines connected to the system
02	Add a new folder to be monitored by the tool in a specific machine
03	Remove a folder from being monitored by the tool in a specific machine
04	List all the folders being monitored by the tool
05	Detect the addition, removal and modification of a file or folder, inside a folder being monitored and automatically update that information
06	Process all the MXF files inside a folder being monitored and extract desired information from them
07	List all the files inside folders being monitored
08	Show detailed information about each of the files
09	Filter files according to its detailed information
10	View replicated files in the system

3.1.1.2 Non functional requirements

The non functional requirements are those that are not reflected in specific features or actions of the tool, but are extremely relevant for the system's behaviour as a whole, and for its architecture.

- **Scalability** — This tool should be able to grow according to the system's size, easily adapting to large volumes of data.
- **Modularity** — This tool should be built so that any module could be replaced and the global solution would still work.
- **Portability** — This tool should be able to adapt to different devices and screens, and work properly in different environments.
- **Performance** — This tool should have a good performance, being able to process large amounts of MXF files in a timely manner .
- **Usability** — This tool should have an adequate user interface for the environment in which it will be used, being user-friendly and efficient.

3.1.2 MATT Workflow

The high level overview of the MATT system is represented in figure 3.1.

The tool monitors a set of N servers and their storage, and analyzes the MXF files inside directories previously configured by the user for each machine. It provides information about the servers and its files to a web application.

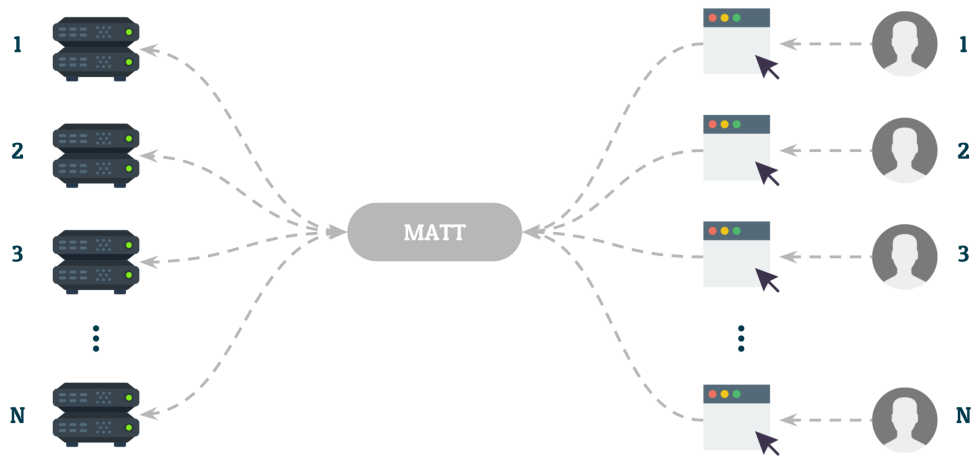

Figure 3.1: High level system overview

The users can access the web application through a web browser. When the webpage is loaded for the first time, the user sees the list of active machines in the system and is able to configure the folders to be monitored in each machine. After the addition of folders for monitoring, the MXF files inside those folders are processed and detailed information about them is collected. If a file or folder is added, removed or modified inside a folder being monitored, the application immediately and automatically updates the information accordingly.

In the web application, the user is also able to view a list of all the folders being monitored and to add new folders or remove active ones. It is possible to view all the files processed and the detailed information about them. The user can filter files according to that information and list all the replicated files in the system.

3.2 System Architecture

After understanding exactly what it is expected from the tool, it is now possible to start thinking how to achieve these goals and which may be the best choices for the development of this solution.

Taking into consideration the solution's requirements and features, it was possible to achieve the system's architecture described in figure 3.2. This architecture is fully capable of performing all the tasks expected and described in the solution's functional requirements, and at the same time fulfill all the non functional requirements, guaranteeing it is a long term solution.

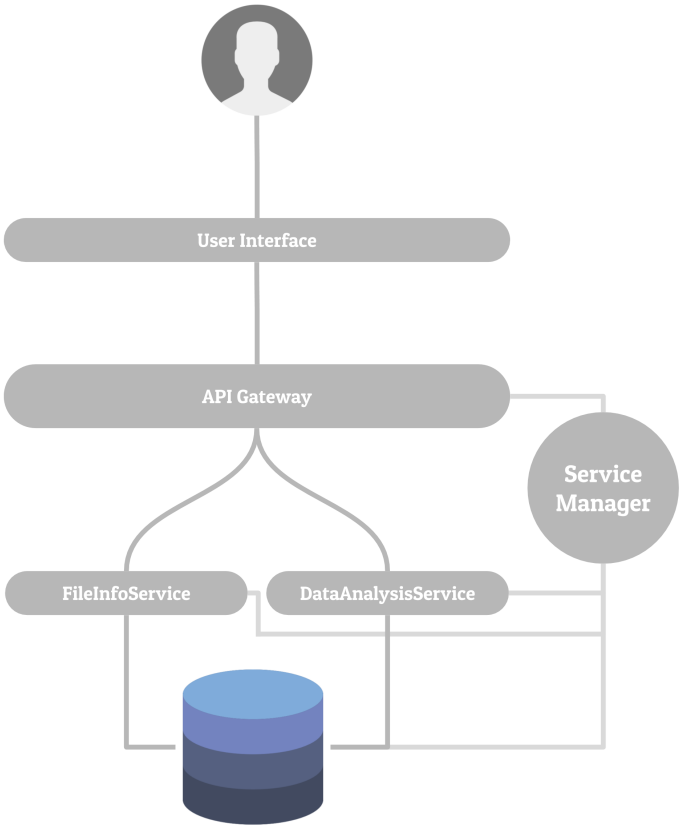

Figure 3.2: MATT system architecture

Once two of the non functional requirements are scalability and modularity, having a microservices architecture seemed a clever decision. This approach will allow to scale the solution as it grows, with more instances of the services, and the development of new ones. The services communicate with each other using REST. There is an instance of the FileInfoService per each machine in the network and as many instances of DataAnalysisService as needed for an efficient performance of the web application. The information will be shared through a database.

3.2.1 FileInfoService

FileInfoService is the service responsible for monitoring an individual machine and its folders, and process and analyze the MXF files. Therefore, it is expected that per each machine, there is an instance of this service running locally.

This service has a list of folders to monitor. Folders may be added to or removed from that list by the user. To do so, the service exposes a RESTful API through which it can communicate with the API Gateway and consequently with the user.

The service watches that set of folders for the creation, deletion or modification of files and folders. It processes all the MXF files it finds inside those folders, extracting a defined set of information from their metadata. There were 17 metadata items identified as relevant for this tool, and these are described in the following table.

Table 3.2: MXF metadata items retrieved from FileInfoService [8]

Name	Meaning
Material Package	Defines the Material Package Set
Source Package	Defines the Source Package Set
Operational Pattern	Specifies levels of file complexity
Start Timecode	The timecode within a track at the starting point of the Essence
Rounded Timecode	Nearest integer frames per second
Length	Timecode set length
Drop Frame	Specifies whether timecode is drop frame or not
Frame Layout	Specifies the frame layout (interlaced, single frame, full frame, etc.)
Frame Rate	Specifies the number of addressable elements of essence data per second
Display Height	Specifies the height of the presented image relative to the sampled image in pixels
Display Width	Specifies the width of the presented image in pixels
Stored Height	Specifies the integer height of the stored image in pixels
Stored Width	Specifies the integer width of the stored image in pixels
Audio Sampling Rate	The reference sampling clock frequency as a rational number
Channel Count	Number of sound channels
Audio Tracks	Number of sound tracks
Bit Depth	Number of quantization bits

After collecting this information from each file, the service adds that data to the database so that it can be used by other services. If a file is created inside a folder being monitored, the service automatically detects it, processes the file and adds the information to the database. The opposite happens if a file is deleted. If a file is modified, the service reprocesses it and the database is updated. This way, the database always contains a perfect mirror of each machine.

3.2.2 DataAnalysisService

DataAnalysisService main responsibility is to retrieve ready to consume data to the user interface. To do so, it exposes a RESTful API which allows it to communicate with the API Gateway.

When the user interface asks for specific information to the API Gateway, the API Gateway will make a REST request to the DataAnalysisService asking for that data. The API of this service receives the request and the service establishes a connection with the database, querying all the data it needs. Then, if needed, the service processes the information to organize it as expected by the user interface, and sends it back to the API Gateway. The response is structured as a JSON object.

There might be several instances of this server located in several machines. If there is a high number of requests, to the API gateway, the system will have a better performance with various instances of this service answering to those requests.

3.2.3 Database

The database is a crucial element of this architecture. It is used to store all the information about the files and the servers being analyzed and it is constantly being asked for information to present to the user. Therefore, it must be carefully chosen and designed, so that it will not have a negative impact on the system.

One of the non functional requirements of this tool was its scalability, and the database is a sensible part of the solution regarding it. Some databases are extremely difficult to scale when systems become more complex, and their performance really decreases, making the system very difficult to use. As described in section 2.3.3.1, Apache Cassandra is an extremely scalable and performant solution, and therefore it was chosen to be the database in this architecture.

At this prototype stage, the database is running with a single cluster, in a single machine, but scaling it to more machines and clusters should not be complex as Cassandra automatically manages its clusters and the information contained in each of them.

The database should be running at a point known by all the FileInfoService instances so that they can add information when they process an MXF file. At the same time, all the DataAnalysisService instances should be able to access the database to extract and process the information about different machines and files. Additionally, the Service Manager should also be able to access the database in order to delete information regarding a machine in case that machine goes down unexpectedly.

3.2.4 API Gateway

The API Gateway is the element that connects the user interface to the middleware layer, allowing the UI to access the information contained in the database, and have interaction with the different machines in the network.

Its main responsibility is to dispatch requests made by the user interface. To do so, it has to have a global vision of the system, understanding which services are up or down and where are these services running. It relies on the service manager and the service register, which help accomplishing this goal.

In section 2.3.1.1, two methods of service discovery were presented and described. In this solution, the server-side discovery approach was used, therefore, when a request is made to the API gateway, it has to decide which service should reply to that request, discover where it is located and dispatch the request. To keep track of the services, the API gateway has a local database that keeps all the needed information. This database is known as the *service registry*.

The API gateway's location must always be known by the user interface, once all the requests that the UI makes have to go through it. It must also be able to connect to all the services registered in the service registry, so that it can dispatch the requests and get the desired replies.

The API gateway may also be responsible for other tasks such as cross origin resource sharing, security and authentication, rate limiting, logging or data aggregation. However, in this prototype stage, the majority of these features are still not implemented. Nevertheless, as described in section

2.3.1.1, Kong is a plugin based solution that allows the addition of a plugin on the top of the API gateway at anytime, without changing the remaining configuration of the gateway. This means that additional features for the API gateway can easily be integrated in the future.

3.2.5 Service Manager

The service manager has the responsibility to monitor and orchestrate all the services comprised in the solution. It is deeply connected with the service registry, being the only service that updates information in it.

When a service is started, it should notify the service manager, informing it where it is running. The service manager will then update that information on the service registry.

The service manager periodically performs health checks to all the active services registered on the service registry. It does a simple call to each service's API and expects an *"I'm alive"* response. If that answer does not arrive, the service manager will consider that service as inactive and will deregister it from the service registry.

All the services must know the service manager's location so that they can register themselves. The service manager should be able to communicate directly with all the services to check their status.

3.2.6 User Interface

The user interface is the module responsible for directly interacting with the user. It is a web application that communicates with the user, presenting him information and performing actions triggered by him.

As described in section 3.1.2, the application is expected to have different views for different information to be presented, as well as all the elements that guarantee that different features of the system work.

The UI always makes its requests to the API gateway through REST commands. It expects data coming from the API gateway structured in a JSON object and ready to be consumed and presented to the user.

3.2.6.1 Mockups

To study the usability of the system and to have guidelines for the development of the UI, several mockups were designed. These mockups can be found in the appendix B.

The mockups were inspired in a regular file system of an operating system. The user can easily navigate through the machines, its folders, and the files inside those folders. This approach will allow the user to feel he is remotely accessing the machine, once he will be presented a real time replica of the folders and files that the different machines contain. To make an user-friendly application, there was an effort to keep the visual aspect of the solution as simple and clean as possible, choosing light colors and fonts. This way, the usability of the application is guaranteed

once it will not be confusing and difficult to use. However, all these visual characteristics may be easily changed in the future.

To help visualizing the final solution, a tool named *Invision* was used to animate the mockups and link them, creating a non functional prototype. This prototype was pretty close the final functional solution, and allowed the analysis of the different elements and views, and the interaction between them.

In all the views that were designed, it was projected the implementation of a *dashboard* in which a global report, with relevant statistical information about the system may be found. Besides this feature, it was also projected the implementation of a *settings* view, that will allow the user to configure its personal preferences and data. However, these two views were not implemented on this prototype stage.

3.3 Conclusion

Assembling the previously described modules, a global and efficient solution can be achieved. This solution not only fulfills all the functional requirements, but also respects the non functional ones.

The responsibilities are spread among different services, and each service performs simple tasks, respecting two important principles of the service oriented architecture. If one service fails, that does not compromise the global application.

The solution is ready to be installed on the cloud. New modules can easily be added, improving the system complexity, and current modules can be replaced or reused in other projects. The database is extremely scalable and capable of keeping up with more complex systems and higher number of requests. The API gateway is capable of dispatching the requests and balancing their load. At the same time, new features can be easily added on the top of the current API gateway. There is an user interface, with special focus on the usability.

In general, the big picture of this solution is well defined. This process is extremely important because it describes each modules responsibilities and works as guidelines for the development stage.

Chapter 4

System Implementation

In this chapter all the details regarding the development of the solution will be presented. Firstly an introduction will be made, explaining how the development stage was organized, and after that, each module's implementation will be described in detail.

4.1 Introduction

Analyzing chapter 3, one can conclude that the system architecture is quite complex comprising various modules with different tasks and responsibilities, and therefore resorting to several different technologies. This is one of the reasons why it is important to have a well defined architecture before the development stage, otherwise the development could become an extremely confusing task.

Once in total there were six modules to be developed, it was necessary to understand in which chronological order did it make more sense to develop them, so that the different modules could be gradually integrated with each other.

The first module to be developed was FileInfoService. This happened because it seemed logic that first of all, it would be necessary to detect and process the MXF files, otherwise the rest of the system would not have data to work with. After that, the database was designed and set up. The FileInfoService was also configured so that it would be able to populate the database. With a populated database, it would be easier to develop the DataAnalysisService, and for this reason this was the third module to be implemented. With both services ready, the API gateway was installed and set up. After that, the service manager was implemented to automate the registration and deregistration of the services on the API gateway. Finally, the user interface was developed and the prototype was concluded.

4.1.1 Methodology

Developing all the different modules in such a tight schedule was quite a challenge. It was only possible with some previous time planning and with a strategy outlined.

Even with the work plan previously outlined, it was used an agile approach instead of the classic waterfall one. This means that the development was an iterative process with constant feedback coming from regular meetings as the solution was growing. This feedback had immediate influence on the work being done guaranteeing that the prototype was being developed as expected. Even though no formal methodologies were used to manage the development of the project, there were weekly meetings with the company's team to present the results achieved in the previous week and plan the work for the following week. Whenever a technical decision had to be taken, there was a presentation about the problem and the different solutions. These solutions would be discussed by the MediaGap's engineering team so that the decision would be taken according to the company's strategy.

All the sub-projects were developed resorting to the company's resources. The code was kept in the company's *BitBucket* repository. It was granted access to the project management tools used in the company such as *Atlassian Jira* or *Atlassian Confluence*. Besides this, all the *DevOps* and testing tools developed and used at MediaGaps were made available to support the development of MATT.

These mechanisms were extremely helpful in the development stage of this tool. This constant evaluation and reflection over the developed work, allowed a critical view over the solution and an adaptation towards the desired product, never loosing sense of time, so that the deadlines could be met.

4.2 Development

The information comprised in the two previous chapters of this document, alongside with the work plan and strategy previously mentioned, create a solid basis for the development of the solution.

In this section, all the technical challenges for each of the modules will be described.

4.2.1 FileInfoService

FileInfoService is the most complex module of the entire solution. It is responsible for monitoring a set of folders, detecting the creation, deletion or modification of a folder or file, processing all the MXF files and inserting data in the Cassandra database.

To achieve these goals, the different tasks were split in two sub-projects, one of them, named *HotFolder*, responsible for monitoring the folders and looking for modification or new folders or files. The other, named *MxfFileInfo*, responsible for processing the MXF files and extracting the desired information from them.

Both projects were exclusively developed using C++, which , as explained in section 2.3.4, is an extremely performant language for low level tasks like these. These C++ projects are then wrapped in another project, named *FileInfoService*, which assembles the two sub projects together, creates the RESTful API and inserts the data in the database.

It is important to understand the concept of *watcher* and *worker*, to figure out how these projects were thought and developed.

The *watcher* is an entity that is responsible for watching a set of folders and wait for some events to happen. In this case, these events are the addition, removal or modification of a file or a folder. Therefore, the *watcher* is closely connected with the *HotFolder* project.

The *worker* is an application that performs a certain task when it is invoked. In the current solution, the *worker* is processing and analyzing MXF files, but in other solutions it may be handling other formats and performing different tasks. Therefore it is closely related to the *MxfFileInfo*.

These two concepts provide modularity to the system, because they are agnostic to each other and can be replaced or used in other solutions.

All the previously mentioned projects will be individually explained and described.

4.2.1.1 HotFolder

The HotFolder is a C++ project that watches a set of folders, looking for the addition, removal or modification of folders or files. Therefore, it is the *watcher* of the current service. When a file is found, it invokes an external *worker* to perform a certain task with that file. The *HotFolder* system is agnostic to the task performed by the *worker* and this should be passed as an argument when the *HotFolder* project is invoked.

To the outside world, the *HotFolder* project exposes two public methods that allow the addition and the removal of a folder to the set of folders the solution is at that time watching, as described in figure 4.1.

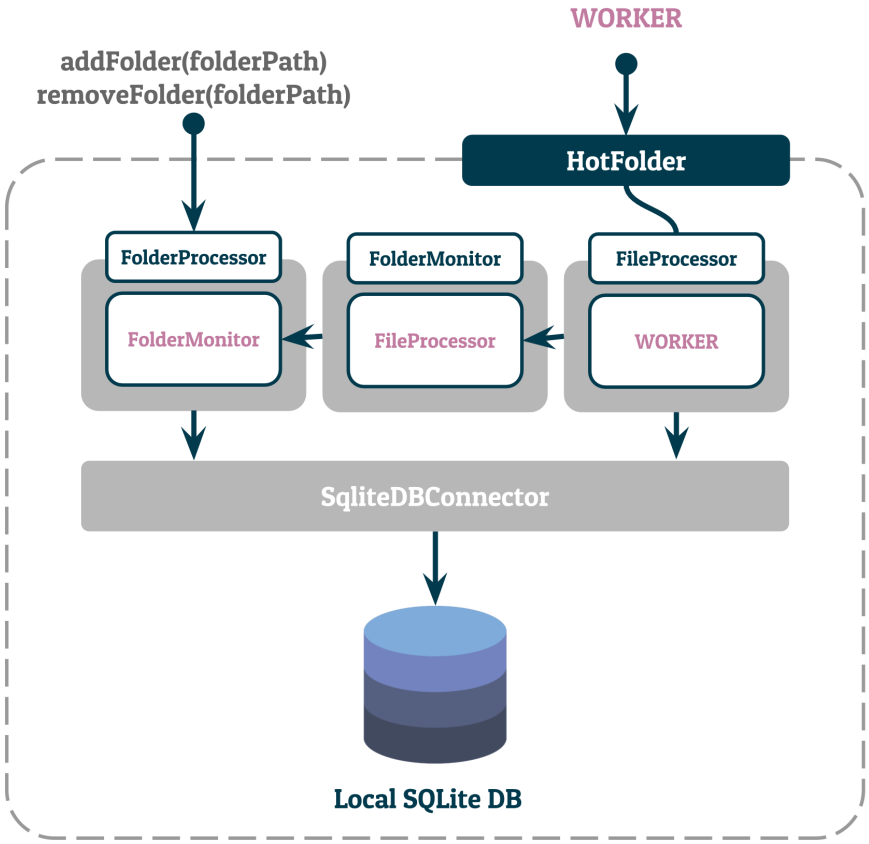

Figure 4.1: HotFolder Project's Architecture

The project is based in three main classes. Two of them, *FolderProcessor* and *FileProcessor*, are similar and derive from the same interface. The *FileProcessor* is used every time a new file is discovered and has to be processed, and *FolderProcessor* does the same for folders.

There is also a local SQLite database, that keeps track of the folders being monitored by the service, and the files that have already been processed. This database is important in case of a system failure, because it will allow the system to recover knowing exactly what files were processed without needing to reprocess them. To communicate with this database, the *FileProcessor* and the *FolderProcessor* use a class named *SqliteDBConnector*, which does all the queries to the database. To do so, it uses an external driver provided by SQLite, specially developed for the communication between C++ applications and SQLite databases.

The *FileProcessor* class invokes the external worker to process the file that has been discovered. If the external worker processes the file successfully, the *FileProcessor* updates the local SQLite database with that information.

Another main class of this project is called *FolderMonitor*. This class is responsible for the actual monitoring of each folder. It uses an external library named *POCO*, that triggers a different handler every time a a folder or file is created, deleted or modified inside the folder being monitored. In case of a file, it uses the *FileProcessor* object to process that item, and in case of a folder,

it uses the *FolderProcessor*.

The *FolderProcessor* is responsible for managing the different folders being monitored. For each of them, it creates a *FolderMonitor* object. It exposes two public methods that allow the addition and removal of new folders. When the system starts up, it checks the local database to see if any folder has been previously added for monitoring and if so, it starts monitoring them again. Everytime a new folder is discover by the *FolderMonitor*, this information is also updated in the local database by the *FolderProcessor*.

4.2.1.2 MxfFileInfo

This project's goal is to process a given MXF file and extract a set of items from its metadata. Therefore, this is the *worker* of the current service.

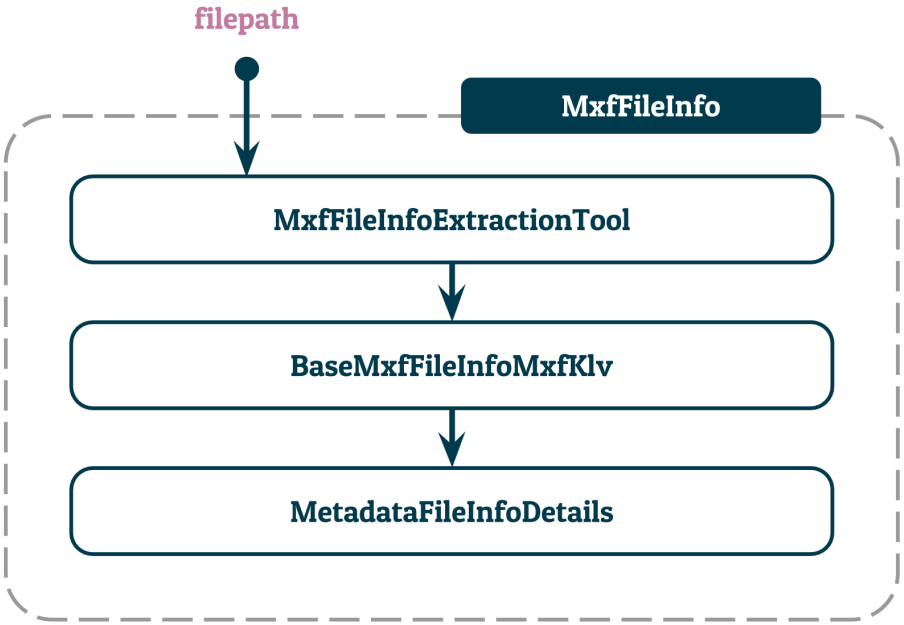

Figure 4.2: MxfFileInfo Project's Architecture

To do so, the project is structured in three main classes. The first one is called *MxfFileInfoExtractionTool*, and gets as input the file path of the file to process. Then it opens the file and goes through it looking exclusively for metadata packages. To do so, it relies on an external library named *EbuLibMxf* developed by EBU, specially to analyze MXF files.

When a metadata package is found, the class *BaseMxfFileInfoMxfKlv* is invoked. As mentioned in section 2.2.2, the MXF file is structured in a key-length-value structure. In this class, the key of the founded metadata package is compared to a list of keys of the 17 items the system is looking for. If the key matches with one of the keys in the list, the value of that metadata package is loaded to a *MetadataFileInfoDetails* object.

The *MetadataFileInfoDetails* is a structure ready to load all the 17 metadata items selected, and return them when the file is processed.

4.2.1.3 FileInfoService

FileInfoService assembles the two previous projects. It wraps both of them, making sure they work together as expected. To do so, it has two layers. The lower layer is developed in C++ and comprises the *HotFolder* and the *MxfFileInfo* projects. The upper layer is developed in NodeJS and allows the integration with the rest of the system and the communication with other services.

The integration between both layers is done using *node-gyp*. *node-gyp* is a tool that allows the compilation of C++ modules as NodeJS Addons, allowing them to be used in a Node JS project just as any other Node JS module.

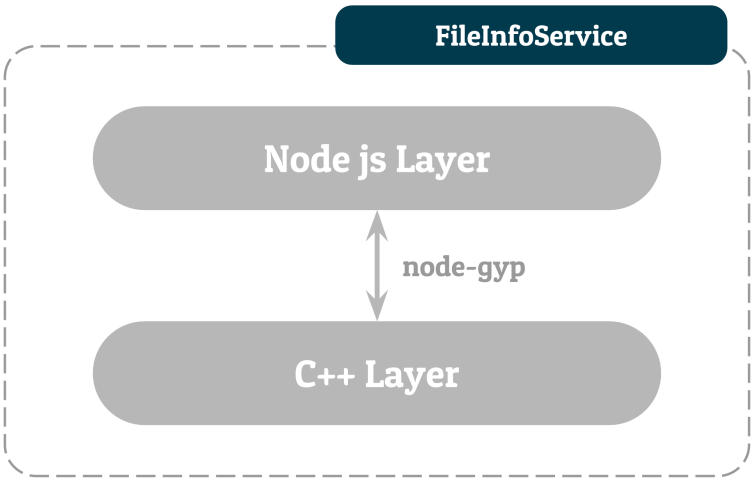

Figure 4.3: FileInfoService Project Layers

Looking to the project's architecture in detail, it is possible to understand exactly how each layer is structured.

Analyzing the NodeJS layer, one can conclude that this works as a RESTful API to communicate with the outside world, receiving requests from the other services, namely the API gateway. To develop this API, a Node JS module called *Express* is used, which makes the service listen for HTTP requests and, handle those requests accordingly. Whenever an user wants to add or remove a folder from monitoring, this API receives a HTTP POST request with the path to the folder to add or remove as a parameter. This request will trigger the addition or removal of those folders. As previously mentioned, it is possible to use C++ projects as NodeJS modules using *node-gyp*. Therefore, when the API receives a request to add or remove a folder, it will call a C++ method from the HotFolder project.

The API is also responsible for registering the service in the service register. To do so, when the service starts, it sends a HTTP POST request to the service manager informing it it is running and what is its location. To make the request, it uses a NodeJS package named *request*, which allows making different HTTP requests, configuring all its parameters. The API uses a class named *ServiceRegistration* to perform this operation.

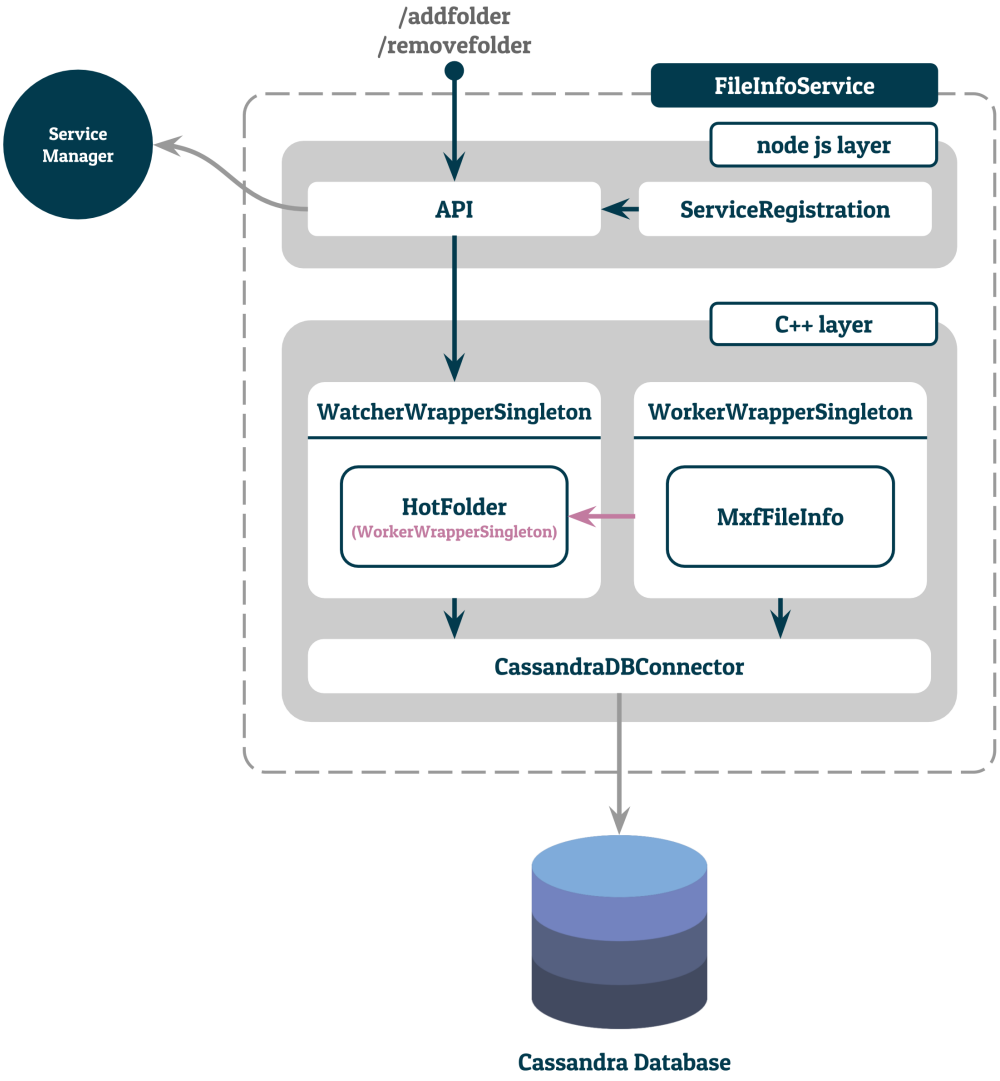

Figure 4.4: FileInfoService Project's Architecture

In the C++ layer there are also two main classes that represent the *watcher* and the *worker* entities, previously described. Each of these classes wrap one of the C++ projects mentioned in subsections 4.2.1.1 and 4.2.1.2. In this wrapping process it was used the *Singleton* design pattern, which guarantees that there is only one instance of each object *watcher* and *worker*. This solution was needed to make sure that every time a new folder was created or deleted, it was not created a new instance of the watcher or the worker. In other words, every time an user is adding a new folder for monitoring, the same object will be accessed.

The *WorkerWrapperSingleton* wraps the *MxfFileInfo* project. Besides that it also adds the information to the global Cassandra database every time a file has been successfully processed. This object is passed to the *WatcherWrapperSingleton* which wraps and coordinates the *HotFolder* project. The *HotFolder* uses the *WorkerWrapperSingleton* as the external *worker*, passed as input.

The *WatcherWrapperSingleton* updates the global Cassandra database every time there is a change in the list of folders under monitoring.

To access the Cassandra database, there is a class named *CassandraDBConnector*. This class uses an official driver developed by *DataStax*, specially developed to enable the communication between C++ modules and Cassandra databases.

4.2.2 DataAnalysisService

The *DataAnalysisService* is responsible for extracting and retrieving data from the database to the API gateway. It should also retrieve the data structured so that it gets to the user interface ready to be consumed, without any additional transformation needed.

This service is exclusively developed in NodeJS, and has 3 classes as represented in figure 4.5.

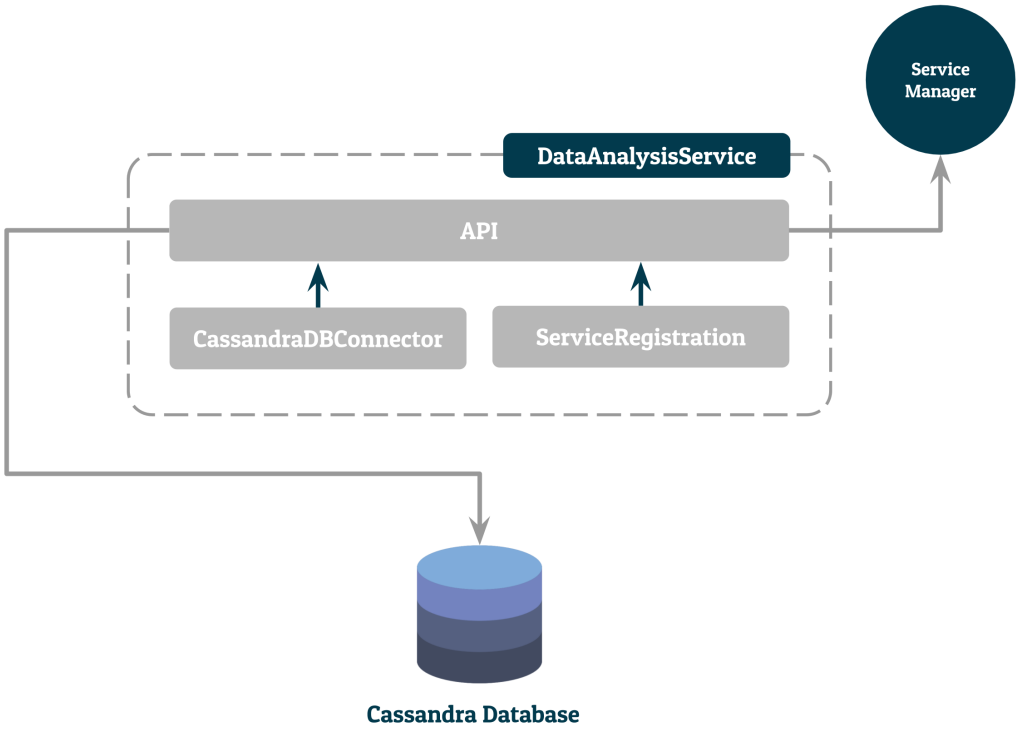

Figure 4.5: DataAnalysisService Architecture

To receive the requests coming from the API gateway, the service exposes a RESTful API to the outside world. Whenever a request is received, this API handles the request and retrieves the desired data.

The service is ready to answer to 12 REST requests, described in table 4.1.

Table 4.1: REST commands exposed by *DataAnalysisService*

Name	Meaning
getAllMachines	Retrieves all the active machines
getAllFolders	Retrieves all the folders being currently monitored
getAllFiles	Retrieves all the processed files
getAllFileDetails	Retrieves all the information about all the processed files
getFilterResults	Retrieves the list of files filtered according to various parameters passed as arguments in the HTTP request
getReplicatedFiles	Retrieves the list of replicated processed files
getFileInfo	Retrieves all the detailed information about a specific file
getAllFilesByMachine	Retrieves all files grouped by machine
getAllFoldersByMachine	Retrieves all folders being monitored grouped by machine
getAllFilesByFolder	Retrieves all files grouped by folder
getAllFoldersInMachine	Retrieves all folders in a specific machine passed as a parameter
getAllFlesInFolder	Retrieves all files in a specific folder passed as a parameter

To develop this API, a Node JS package called *Express* is used, which makes the service listen for HTTP requests and, handle those requests accordingly. All of them are HTTP GET requests, and some of them need to contain some parameters that are imperative for information to be retrieved. To get the data from the database, the API uses a class, named CassandraDBConnector. This class uses a Node JS package named *cassandra-driver*, developed by *DataStax*. According to the request, this class builds and executes a query to the database. After data is retrieved from the database, it is structured in a JSON object so that it is ready to be used in the User Interface.

The class named *ServiceRegistration* helps the API in the registering process, sending a HTTP POST request to the service manager informing it that it is starting and running at its location. To make the request, it uses a Node JS package named *request* that allows sending different types of HTTP requests.

4.2.3 Database

As mentioned in section 3.2.3, the Apache Cassandra database was chosen to integrate this architecture, due to its high performance and scalability.

In Cassandra, some concepts are quite different from what is normally expected in a traditional relational database. Therefore, it is important to have these concepts in mind while designing and integrating a solution with a Cassandra database. In this kind of databases, it is normal, and even encouraged, that information is replicated over several tables. A developer should not be afraid of writing several times the same information, in different tables. This happens not only because writes are extremely cheap, meaning they can be performed almost instantaneously, but also, because data should be structured based on the way it is going to be consumed. Logical operations are never done on the database side. In other words, operations like joining tables are not even supported. When an user reads something from a database it should come from a simple

and direct query. When designing a Cassandra database it is also important to spread data evenly around different clusters.

After realizing which concepts are important in a Cassandra database, it was necessary to understand how is the user interface going to consume data so that the database can be designed in an efficient way. After some consideration, five different tables were identified as necessary in this prototype stage.

- **machines** — This table keeps the information about the current active machines. To do so, it only needs to have the combination of the *machineIp* and the *machinePort*.
- **folderMonitoring** — This table has the list of all the folders being monitored. It stores the *folderPath*, the *machineIp* and the *machinePort* of each folder.
- **processedFiles** — This table contains the list of all the files that have been processed by the system. Besides the *machineIp*, the *machinePort*, the *filePath* and the *fileName*, it also has the *rootFolder*, which is the folder being monitored that contains that file.
- **fileDetails** — This table has all the metadata information extracted by the solution. Additionally to the five columns necessary to identify the file (*machineIp*, *machinePort*, *filePath*, *fileName* and *rootFolder*), it also contains seventeen more columns corresponding to the seventeen items currently being extracted from the MXF files.
- **fileReplication** — This table is responsible for keeping track of the file replication in the system. It counts the number of files with the same *sourcePackage*. Therefore it is simply the association of a specific *sourcePackage* with an integer counter.

All these tables are contained in the same keyspace, which was named *FileInfoService*. This keyspace was configured with the SimpleStrategy replication strategy and with a replication factor of three, which means that data will be replicated three times on multiple nodes.

In this prototype stage, the database was installed in a single cluster, but it is recommended that in production environments more nodes are used.

4.2.4 API Gateway

Kong was the option chosen to work as an API gateway for this solution. As explained in section 4.2.4, this is an open-source and ready to use application, and therefore, the work to be done regarding this module is more related with guaranteeing that all the requirements for Kong to work are fulfilled, installing it, and configuring it.

Kong offers different installation possibilities, regarding the environment in which it will be running. Generally, most of the installation work is already done, but in some cases it may include things such as installing a local Cassandra or PostgreSQL database, which work as the Service Registry, as described in subsection 2.3.1.1. All the details about the development environment used in the different stages will be explained in section 4.3.

After successfully installing Kong, it is necessary to configure it according to the desired usage. Kong uses two TCP ports to communicate with the outside world. By default, port 8000 is used for HTTP traffic by different clients, while port 8001 is used to access the configuration RESTful API that allows registering or deregistering new services and APIs, enabling plugins, or other configuration tasks.

In MATT, the API gateway is expected to be capable of handling the twelve REST requests described in table 4.1, that should be dispatched to an instance of *DataAnalysisService*. Besides these twelve requests, it should also be capable of handling the requests */addFolder* and */removeFolder*, for each instance of a *FileInfoService*.

For the requests that are dispatched to *DataAnalysisService*, all the twelve APIs are manually configured through the Admin API provided by Kong. For each of these APIs, Kong is informed that it should dispatch the request to an *upstream object* named *dataanalysis.service*. This *upstream object* is a list of all the current instances of a service. Each instance is called a *target*, and whenever a request for the *DataAnalysisService* hits Kong, it will redirect the request to one of the *targets* of the *upstream* list. Each of these targets has a parameter, named *weight*, and therefore, by default, Kong will redirect the request to one of the *targets* in the *upstream* using the round-robin algorithm.

For the requests that are dispatched to *FileInfoService*, the two APIs are configured automatically by the Service Manager, for each instance of the service. This happens because when one of these requests is dispatched it should reach a specific machine, running at a specific location, once it is in that machine that the user wants to add or remove a folder.

Finally, it is necessary to configure one plugin in order for the API gateway to work properly. This plugin is provided by Kong and it is named CORS (Cross-origin resource sharing). It defines which different origins and methods are allowed to use the API gateway. In this case, all the methods and origins were allowed since it is a prototype and no concerns regarding security were taken. To add this plugin, it was only necessary to make a HTTP POST request to Kong's Admin API, informing that the plugin should be enabled with the previously described configuration.

4.2.5 Service Manager

The Service Manager has a close connection with the API Gateway. It is the entity that registers and deregisters all the APIs and targets. It is therefore crucial to the normal behaviour of the application.

In figure 4.6, it is possible to analyze the structure of this service.

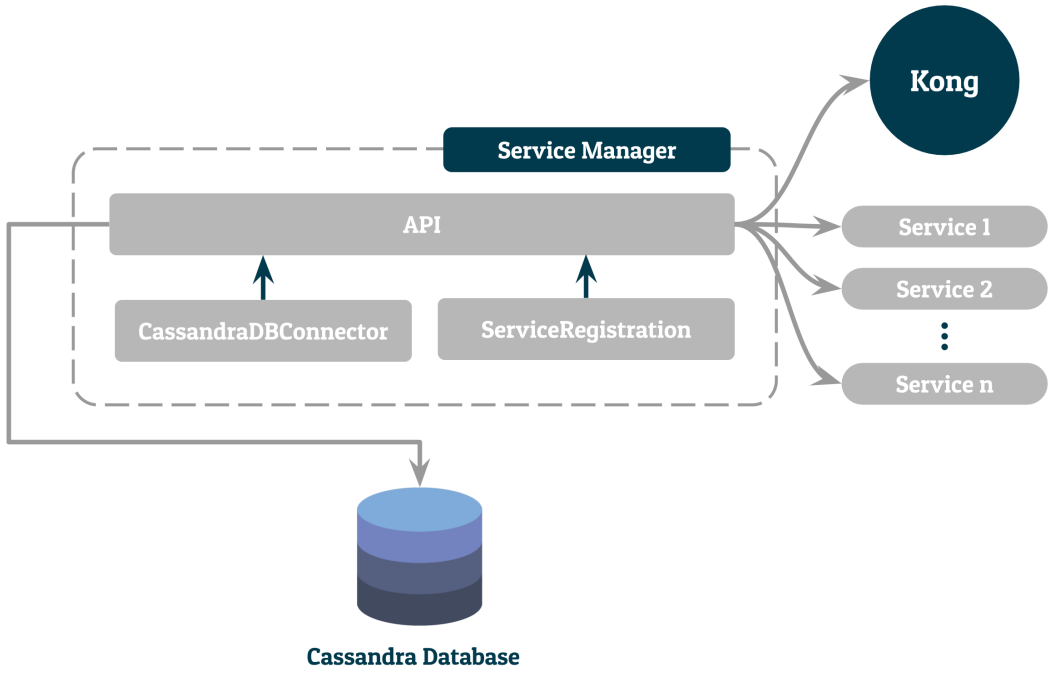

Figure 4.6: ServiceManager Architecture

The service has an API that communicates with the outside world, receiving requests from the different services, and making requests to the API gateway, to register new services running.

The Service Manager is waiting for two REST requests, */registerFileInfoService* and */registerDataAnalysisService*, one for each kind of service. To do so, it also uses the Node JS package *Express*, which listens for these requests and handles both of them. To register these services in the API gateway, the Service Manager uses a NodeJS package named *request*, which allows making different HTTP requests configuring all its parameters.

Besides that, the service manager should ask the API gateway for all the current active services and check if all of these are still running. To do so, it hits all services' APIs with the */health* request, waiting for a successful reply. If that reply never arrives, the service manager informs the API gateway that the service is no longer available, and in case of being a *FileInfoService*, it also accesses the Cassandra database, updating the information about the machine that is no longer running the service. To communicate with all the services' APIs the *request* Node JS package is used again. To access the Cassandra database, the API uses a class in which all queries are prepared. This class, named *CassandraDBConnector*, uses a Node JS package named *cassandra-driver*, developed by *DataStax*.

4.2.6 User Interface

The user interface is the module that directly connects to the user and allows him to take actions on the system. It is closely connected to the API gateway, because all its requests go through it.

The UI was developed using ReactJS, which as described in section 2.3.6.2, allows the creation of a set of views, that will be rendered according to the current state. The views were developed based on the mockups previously designed, as described in section 3.2.6.1.

Whenever the user makes an action to change the view, data needs to be loaded so that the view can be rendered. To do so, the UI does a specific request to the API gateway, which redirects the request to a service and retrieves the information ready to be consumed. Then the information is organized, and the page is finally rendered and presented to the user.

Bootstrap is a HTML, CSS and JS framework that helps creating responsive web applications. To help the development of this user interface, it was used *React-Bootstrap*, which is the same front-end framework, rebuilt for the React environment. This framework is really helpful in the visual organization of the information, simplifying the creation of tables, pop-ups, warnings, and other elements, and facilitating the development of responsive solutions.

4.3 Development Environment and Cloud Setup

In a system with so many different modules, developed in several different technologies, it may not be easy to find a strategy to integrate them, specially in the development stage, when modules are not completely finished, but should start to communicate with each other so that the structure of the solution can evolve towards the final product.

The natural approach while developing the first projects was to use a single machine, and develop modules that would run locally. The Cassandra database was also installed in the same machine and communicated with both *FileInfoService* and *DataAnalysisService* locally. This was enough during the initial times of development, but when complexity increased and it was time to integrate the API gateway in the solution, it was necessary to start considering different solutions. It was easily concluded that some sort of virtual machine would be a simple solution.

Docker is a software container platform, that allows running an application wrapped in a container, which can communicate with other containers, without the painful process of configuring virtual machines. Fortunately, Kong provides a pre-configured Docker container, which means that it was relatively easy to run a Kong in the same machine as the other services. It was only necessary to install an Apache Cassandra container for Kong to use as the Service Registry, then install the Kong container and run both of them. After that, the services, the API gateway and the database could all run and communicate in the same machine.

Finally, the user interface was also installed in the same machine, making all its requests to the API gateway running in the Docker container.

This set up was used during the development stage and allowed the development of the different modules, but it is obviously far from what it is expected in a real life situation. Therefore it was necessary to try to reach a more realistic solution that would allow to have a better perception of how would the system behave.

To reach this goal, the solution was moved to the cloud, which means that, instead of having the different services running locally in the same machine, they started being run in distributed

machines. To do so, Amazon Web Services (AWS) were used to launch three different machines, and the modules were split among them as described in figure 4.7.

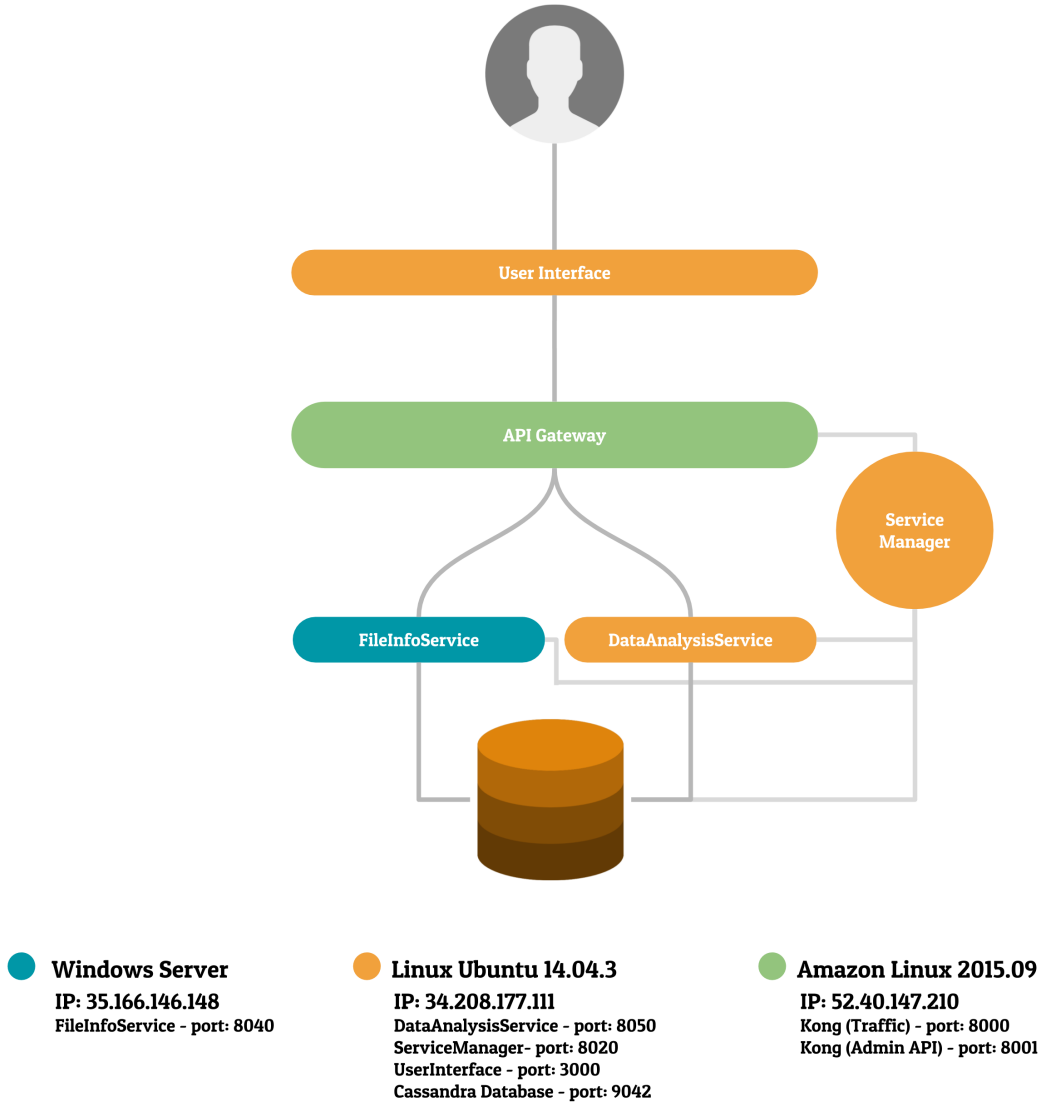

Figure 4.7: AWS Setup

Kong provides a pre-configured and ready to use AWS machine, therefore, to start the API gateway it was only necessary to launch this machine and do the required configuration, as described in section 4.2.4. After that, Kong was ready to start receiving and dispatching requests.

The User Interface, the Service Manager, the DataAnalysisService and the Cassandra Database were deployed in a Linux Ubuntu machine.

Finally, a third machine was set up for FileInfoService. This was a Windows Machine which contained not only the service running, but also some MXF files to test the system.

All the machines had a public IP addresses, which means that they could be reached from the

outside world, and therefore reach each other. As described in figure 4.7, each module has its own TCP port opened to communicate with other services and answer the requests. Each service must know where some of the services are running. For instance, every time a *DataAnalysisService* or *FileInfoService* instance starts running, they should connect to the Service Manager and to the Cassandra Database, which should be running at fixed locations. The same happens when the User Interface needs to make a request to the API Gateway. In this prototype stage, this issue was solved using configuration files. For each service there is a file which indicates the location of the modules that that service needs to know. This way, each service can be easily configured to work properly in different set ups.

4.4 Tests and Performance

When building software with such complex architecture, and integration with different modules, it is important to test the solution as it grows and as a final product, so that it is possible to quantify and measure the quality of the developed work, understanding if it meets the expectations.

Throughout the solution development, there was a concern to test each module behaviour through unit tests. This kind of tests was more intense in the lowest levels of the solution, namely in the development of the *HotFolder* and the *MxfFileInfo* projects. With unit tests it was possible to examine small features of each project and guarantee that those would work, even if new features were implemented. In some cases, it was even used a test-oriented approach in the development of the solution. In other words, the unit test was designed and developed firstly, and only after that the actual code would be written to fit and fulfill that test.

These unit tests were also integrated with Jenkins, the test automation solution used at MediaGaps. Jenkins runs all the tests for a specific project every time there is a new commit of code to the repository. It builds the solution and runs all the tests, registering the test results, and keeping a report of the stability of the solution over time. All these results and reports can be accessed through a web application.

Besides unit tests, some load and performance tests were made, when the solution was integrated and moved to the cloud. The first module to be tested, was FileInfoService. The purpose of this test, was to understand how would this service behave when it had to process a large amount of files in a small amount of time

The test was done with the cloud setup described in section 4.3, which means the modules were distributed among different machines, and the configuration was similar to a real production stage.

The test consisted in adding or removing folders from monitoring. The used folders contained different amounts of files, and therefore, different amounts of data to be processed. Per each different amount, the time to process all the files was measured. All the requests were made through the API Gateway, which then dispatched the request to the specific FileInfoService instance. There were used folders with 1, 10, 50, 100, 250, 500 and 1000 copies of the same MXF file. This MXF

file had the size of 43.5 Mb. Each folder was added and removed three times and in the end the average of the results was calculated.

Table 4.2: FileInfoService Performance Test Results

		Addition Time (ms)				**Removal Time (ms)**			
N Files	**Size (MB)**	**1st**	**2nd**	**3rd**	**Av**	**1st**	**2nd**	**3rd**	**Av**
1	43.5	580	156	625	***454***	134	113	114	***120***
10	435	5439	1305	5278	***4007***	968	744	762	***825***
50	2175	30970	7650	23901	***20840***	4417	3519	3611	***3849***
100	4350	56242	37175	54703	***49373***	7972	7805	7105	***7627***
250	10875	141798	122991	138218	***134336***	20657	21148	20009	***20605***
500	21750	284435	272195	291553	***282728***				
1000	43500	569645	584117	619462	***591075***				

Analyzing the results presented on table 4.2, it is possible to understand that, as the amount of files grows, the time to process them also increases, when a folder is being added. The system is capable of processing approximately 89.76 Mbytes/sec. To process a simple file of 43.5 *Mbytes* it took on average 454 *ms*, and to process 1000 files, which resulted in a total of 43.5 *Gbytes*, it took on average 9 mins and 41 seconds.

When a folder is removed, the processing time is considerably smaller than when a folder is added. This is expected because when a folder is being removed, it is only necessary to delete the information from the database, whereas when a folder is added the process is much more complex, as it is necessary to process each file, and to add the information to the database. However, when deleting a file with the number of files greater than 250, the service crashed and was not able to delete all the files in the database. This may be explained by the high number of queries being made to the database, which was not capable of answering all of them in such a small amount of time. As it is described in section 4.3, in this prototype stage, the Cassandra database is only configured with a single running node. As previously mentioned, for a production stage, which may be capable of handling more demanding tasks, it is highly recommended that the Cassandra database has more than one running node.

As expected, comparing the numbers for the different amounts of files, it is possible to conclude that this service has a linear behaviour, whether a folder is being added or removed. In other words, the time it takes to process 1000 files is approximately 1000 times the time it takes to process one file.

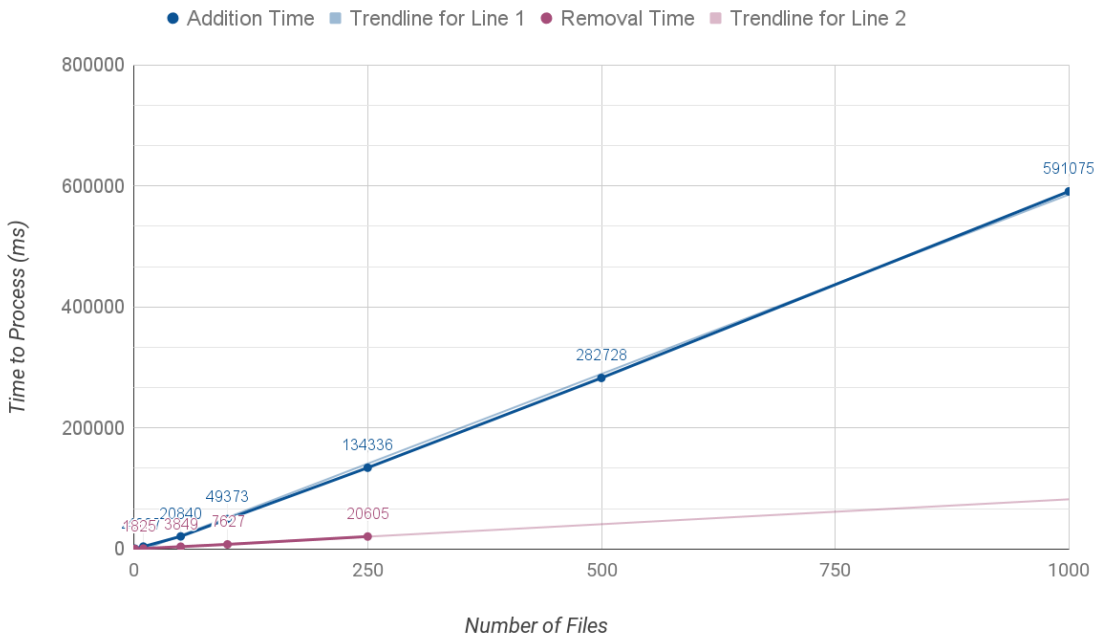

Figure 4.8: FileInfoService Linear Behaviour

After testing the FileInfoService, it was also interesting to do a load test to the whole system, in order to analyze how would it perform under a high number of requests. This test was focused on the API gateway, the DataAnalysisService and the Cassandra database. A variable number of requests was sent to the API gateway, which dispatched them to an instance of DataAnalysisService, which should access the database and retrieve the requested information. Once again, the different modules were running in the cloud, as described in section 4.3.

To do this test, an application named Apache JMeter was used. JMeter allows making a variable number of HTTP requests in a certain period of time. It provides statistical data about the system's response to the requests, allowing to take conclusions about its performance.

In this test, the number of requests made by JMeter, varied among 1, 10, 50, 100, 250, 500 and 1000. These requests were made to the API gateway over the period of one second. The API gateway was responsible for dispatching them to the DataAnalysisService. The request used in this test was */getAllFiles*. The database was populated with 1000 files, therefore it was expected that the response for each request was a JSON object with 1000 elements, containing data about each file. For each of these amounts of requests, the test was repeated with 1, 5 and 15 instances of DataAnalysisService running.

In order to guarantee the test's accuracy, the previously described experience was repeated three times and the average value of the results was calculated. To take conclusions, it was analyzed not only the average response time for each request, but also the percentage of successful replies.

The detailed results of this test can be consulted in appendix A.

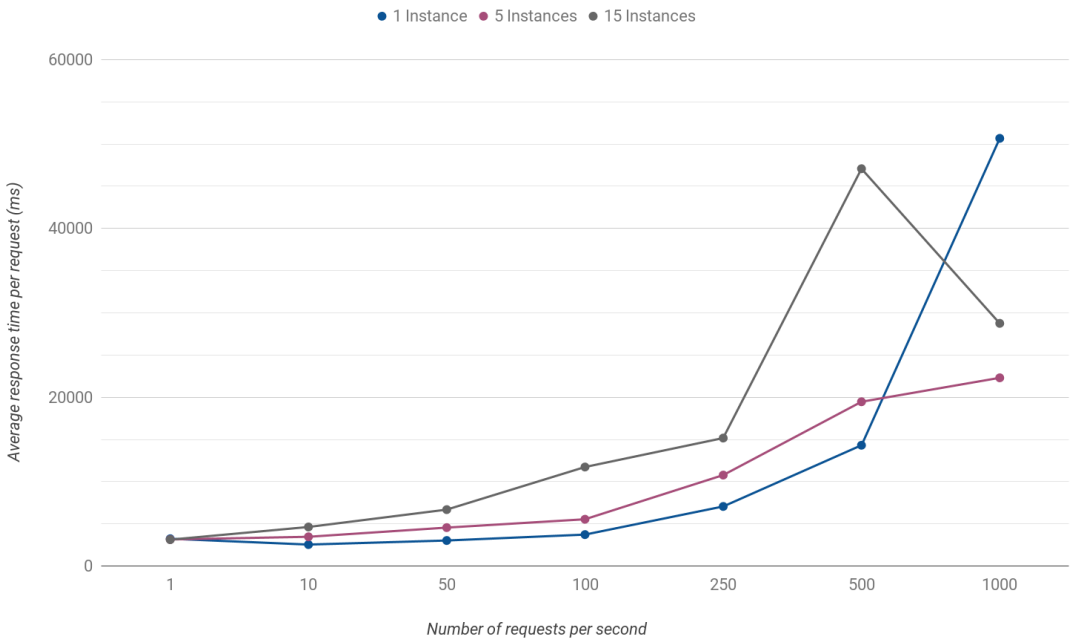

Figure 4.9: DataAnalysisService Average Response Time per Request

Figure 4.9 represents the average response time in milliseconds for each request, for different amounts of requests made per second, and for different amounts of DataAnalysisService instances running. Analyzing this figure is possible to conclude that, as the number of requests increases, the response time also grows. This is an expected behaviour, however, it would also be expected that the system would reply faster to a higher number of requests, with a higher number of instances of DataAnalysisServices running. In fact, the system's behaviour is the exact opposite. The system has a faster response when a smaller number of instances is running.

Figure 4.10 represents the percentage of successful replies for different amounts of requests per second, and for each amount of DataAnalysisService instances. A successful response is defined as one that retrieves the expected JSON object with 1000 elements, and an unsuccessful response is one that retrieves an error message.

Analyzing this figure, one can conclude that the higher the number of DataAnalysisService instances is, the higher the percentage of failure in the system's response will be, as the number of requests per second grows. This is not an expected behaviour as well.

In general, what is seen in both figures is that, for a small number of requests, the system's performance is barely identical, independently of the number of DataAnalysisService instances. When the number of requests per second increases, the system has a better performance with a single instance running than with bigger number of instances.

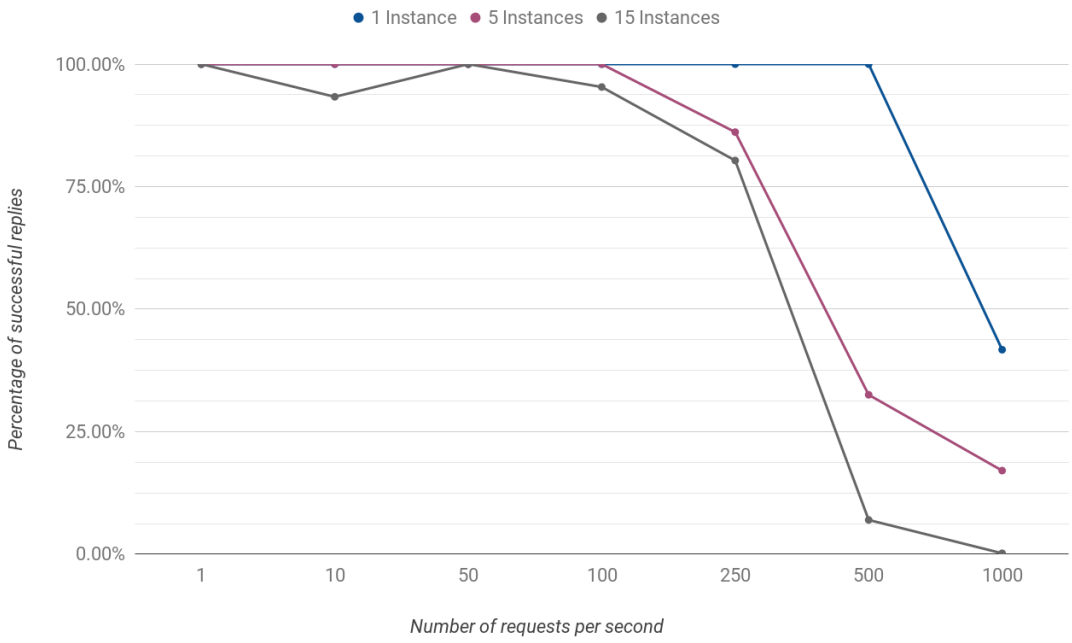

Figure 4.10: DataAnalysisService Percentage of Successful Responses

To discover the reason for these unexpected results, it is relevant to analyze each module's behaviour individually. When the tests were being done, it was possible to observe that the API gateway was having a regular performance, based on the logs of the different DataAnalysisService instances, that showed that the requests were arriving to their APIs. In other words, the API gateway was successfully dispatching the requests among the different DataAnalysisService instances, as expected. The DataAnalysisServices were also behaving normally, because their logs indicate that they were accessing the database as expected. Therefore, one can conclude that the source of problems is at the database layer. The Cassandra database was not able to keep up with request rate, taking too long to reply to a request or even not being capable of answering it.

This happened because, as previously explained, there is a single node of the database running in the current set up, contrary to what is suggested in an extremely demanding environment such as this. Hence, the Cassandra database was not capable of replying to such an amount of requests at the same time, creating a bottleneck and eventually crashing after some time. This is the main reason why the global solution works better with a single DataAnalysisService instance. Theoretically, if there were several nodes answering to requests done by the different instances of the service, the global performance of the system would be increased.

4.5 Challenges and difficulties

Developing all these different modules was, without any doubts, a challenging journey, not only because of the system's complexity, but also because there was a well defined and tight deadline.

First of all, the technical choices that had to be taken, and all the research that supports them, was certainly a demanding task, which required a careful study of the different technologies and their advantages and disadvantages. It was challenging to learn so many different technologies in such a small amount of time, understanding why it would be more indicate to use one of them, for a specific purpose, instead of another one.

Another initial struggle was to understand the usual code organization and structure at MediaGaps, so that the developed solution could easily be integrated in the company's repository, and, at the same time, reuse some code previously developed.

There is a considerable number of external libraries or frameworks used in different parts of this solution. It was not always easy to integrate them with the solution, in a way that the big picture would work as expected.

It was also demanding to move the solution to the cloud, understand how AWS work, and how to correctly launch the different machines required for the tool to work.

Finally, finding suitable tests for the different development stages and technologies of the solution was definitely crucial for the successful delivery of this prototype. However, it was a relatively challenging process as well, as it required some research and deliberation to find tools and methods that would guarantee efficient and relevant tests.

Chapter 5

Conclusions and Future Work

The television production reality is now different from what it used to be a few years ago. In fact, it still is under constant changes, due to the increasing technological development and the pressure imposed by fierce market competition. There is a constant concern to optimize processes, so that the profit can be maximized. This context introduces new challenges in the TV production workflow. In the one hand, there are more and more contents to be managed, stored or broadcasted. On the other hand, there is a new technological scenario, which allows new and more powerful tools to support this kind of tasks.

There is an emergent trend for digital and cloud solutions, based on decentralized, modular and scalable solutions. New coding languages facilitate the development of solutions ready for these environments. If these technologies are combined with traditional and performant languages, it is possible to achieve very interesting results.

This dissertation results from the previously described context and was based in two main components. The first one was a research work about the current state of the art of the TV production industry and the last trends and technologies currently being used in software development. This part was extremely important because it worked as the foundations for the work that was developed afterwards. This work consisted in a functional prototype of MATT. Before the actual development of this tool, all the solution was projected and designed based on the functional and non functional requirements established. Only after this, the different modules of the solution started being developed and integrated with each other. Finally all the performance and load tests were done.

In the end, all this stages were concluded successfully as described in this document, meeting the expectations for this dissertation.

5.1 Goals achievement

The main goal of this dissertation, as described in section 1.2, was to develop a functional prototype of a monitoring tool for TV production systems, named MATT. This goal was successfully

achieved. At the end of this journey, the developed prototype successfully demonstrated to support all of the initially defined requirements, both from a functional viewpoint as well as operational.

In the current stage of the solution, the user is perfectly capable of viewing all the active machines in the system, and easily add or remove folders to be monitored in those machines. He is also able to view all the folders and files, and organize them by machine or folder. It is even possible to filter the files according to their metadata or to view the replicated files in the system.

At the same time, all the non functional requirements were completely respected and fulfilled. One of the main concerns for this prototype, was that it was not a fixed tool. Instead it should be a modular solution, which could easily grow and adapt to several environments. To do so, it was implemented a microservices architecture, splitting responsibilities among different services, which communicate and cooperate with each other. All the technical decisions were taken with special regard to the scalability, modularity and performance of the final tool.

Therefore, the tool developed today is much more than a simple prototype for a tool. Instead it is a well defined and stable framework, constituting a proof that the approach actually works. It can thus be seen as a solid basis for a future commercial product with more complex features and eventually integrated in larger systems.

5.2 Future work

As described in the previous section, the future of this prototype was a concern since the beginning of this dissertation, and, even though all the goals for this thesis were successfully achieved, there is some work that could be developed out of the scope of this dissertation.

First of all, new features can be added to give the user more relevant information about the system and its performance. New modules can be implemented, with special regard to analytics, machine learning and big data algorithms, to eventually predict the system's behaviour and take a step further into the automation of TV production workflows. These new services can be developed using the most convenient technologies, deployed in the most advantageous environment and be easily integrated in the current architecture.

Besides that, a few improvements could be done to the actual prototype. Regarding the set up of this solution, there could be implemented an easier way to configure the communication among different services, instead of the configuration files used in the current version. Regarding the user interface, it would be interesting to review the usability of the different views, and try to adapt them to a closer real life TV production environment. For instance using *Redux*, which is a framework that allows the re-rendering of a current view, every time there is a some data update, without the manual refresh of the page, performed by the user.

In the current solution, it is possible to access any of the modules of the system, ask for information, or even take an action such as adding or removing a folder, without any kind of authentication. No security mechanisms were implemented, however they were taken in consideration throughout the development of the prototype. In other words, all the modules were thought so that it would not be difficult to implement security and authentication mechanisms as the solution

grows. This solutions would probably be built around JSON Web Token (JWT), which allows the secure transmission of information as a JSON object. Once this tool handles big amounts of data, which in most cases is very likely to be confidential, this would be a very important step to bring this prototype closer to the final product version.

Another relevant improvement to this tool, would be the integration with external systems, developed in MediaGaps or even provided by other entities. In concrete, it would be very pertinent for this tool, to have an integration with the EIDR system. This system is an universal unique identifier system. This means that it creates an unique ID for every movie and television assets, so that it is possible to associate an asset with a television or cinema work. It provides an API, that allows the access to a database, where the association between an ID and an asset is made. Therefore it would be possible and interesting to integrate this information in the current solution.

Finally, to turn this tool from a prototype to an actual product that could be commercialized, some research on licensing a product would have to be done, understanding in which way could this be monetized and sold to the market.

Appendix A

Load testing result tables

In this appendix three tables will be presented, containing all the detailed results from the load test described in section 4.4.

[1] *ART* stands for Average Response Time and is measured in milliseconds

Table A.1: Load testing with one DataAnalysisService instance

		1st Test		2nd Test		3rd Test		Av.	
		ART[1](ms)	**Success Rate**	**ART(ms)**	**Success Rate**	**ART(ms)**	**Success Rate**	**ART(ms)**	**Success Rate**
Number of request/sec	**1**	1715	100.00%	3533	100.00%	4491	100.00%	***3246***	***100.00%***
	10	1599	100.00%	3145	100.00%	2923	100.00%	***2556***	***100.00%***
	50	1913	100.00%	3590	100.00%	3611	100.00%	***3038***	***100.00%***
	100	2530	100.00%	4328	100.00%	4352	100.00%	***3737***	***100.00%***
	250	5228	100.00%	8841	100.00%	7146	100.00%	***7072***	***100.00%***
	500	15951	100.00%	13140	100.00%	13881	100.00%	***14324***	***100.00%***
	1000	96079	29.00%	22446	23.50%	33515	72.70%	***50680***	***41.73%***

Table A.2: Load testing with five DataAnalysisService instances

		1st Test		2nd Test		3rd Test		Av.	
		ART (ms)	**Success Rate**	**ART(ms)**	**Success Rate**	**ART(ms)**	**Success Rate**	**ART(ms)**	**Success Rate**
Number of request/sec	**1**	3449	100.00%	3077	100.00%	2953	100.00%	***3160***	***100.00%***
	10	3455	100.00%	3854	100.00%	3128	100.00%	***3479***	***100.00%***
	50	4171	100.00%	3910	100.00%	5618	100.00%	***4566***	***100.00%***
	100	4354	100.00%	4282	100.00%	8025	100.00%	***5554***	***100.00%***
	250	8343	100.00%	8142	100.00%	15880	58.40%	***10788***	***86.13%***
	500	21111	57.20%	20421	13.60%	16894	26.80%	***19475***	***32.53%***
	1000	25104	20.70%	20982	30.10%	20851	0.40%	***22312***	***17.07%***

Table A.3: Load testing with fifteen DataAnalysisService instances

		1st Test		2nd Test		3rd Test		Av.	
		ART (ms)	**Success Rate**	**ART(ms)**	**Success Rate**	**ART(ms)**	**Success Rate**	**ART(ms)**	**Success Rate**
Number of request/sec	**1**	3391	100.00%	3059	100.00%	2966	100.00%	***3139***	***100.00%***
	10	6859	90.00%	3547	90.00%	3515	100.00%	***4640***	***93.33%***
	50	6604	100.00%	7944	100.00%	5544	100.00%	***6697***	***100.00%***
	100	20781	86.00%	7844	100.00%	6606	100.00%	***11744***	***95.33%***
	250	11096	41.00%	18010	100.00%	16422	100.00%	***15176***	***80.33%***
	500	35030	0.00%	58450	10.40%	47770	10.60%	***47083***	***7.00%***
	1000	66806	0.60%	10479	0.00%	9008	0.00%	***28764***	***0.20%***

Appendix B

User Interface Mockups

In this appendix the user interface mockups will be presented. These mockups were designed to guide and help in the development of the user interface, and are a very approximate representation of the actual visual aspect of the developed prototype.

B.1 Dashboard Mockup

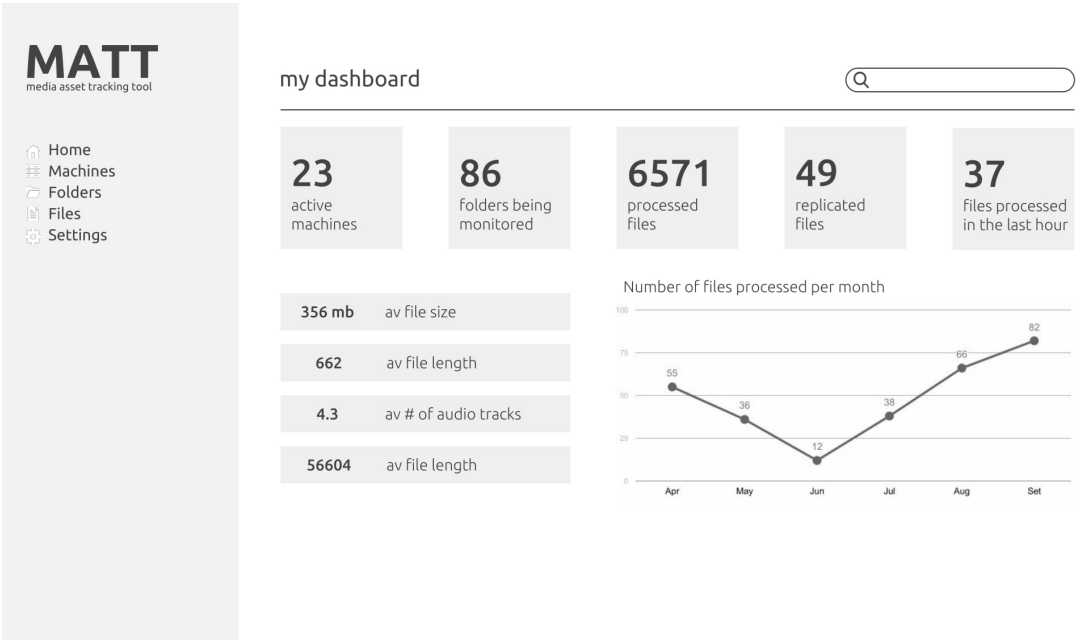

Figure B.1: Dashboard Mockup

B.2 Machines Mockup

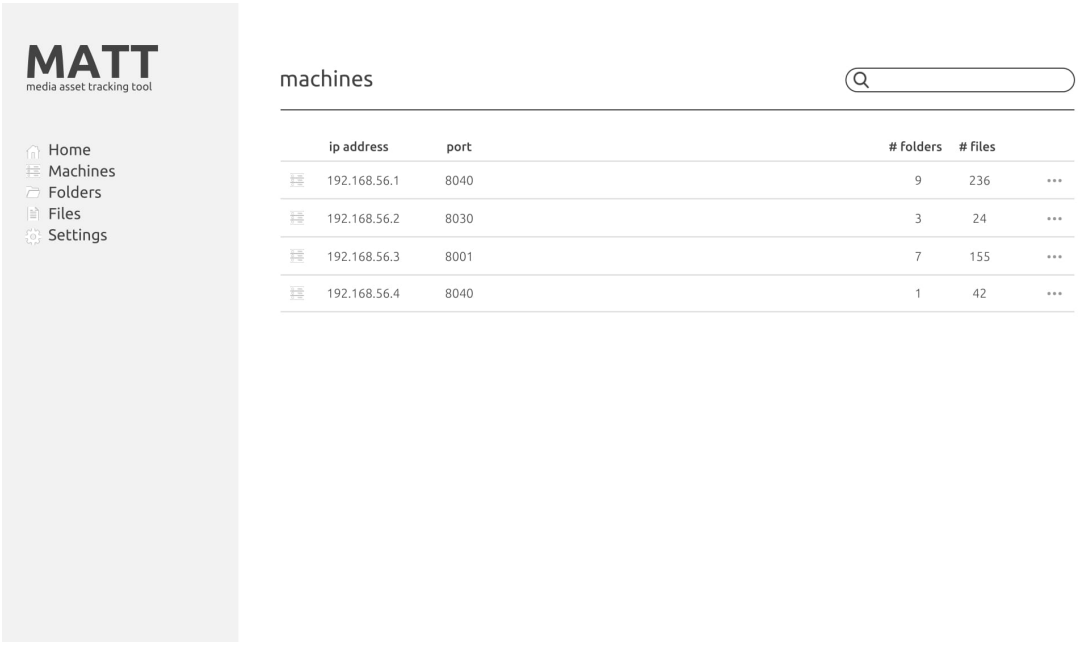

Figure B.2: Machines Mockup

B.3 Machine Details Mockup

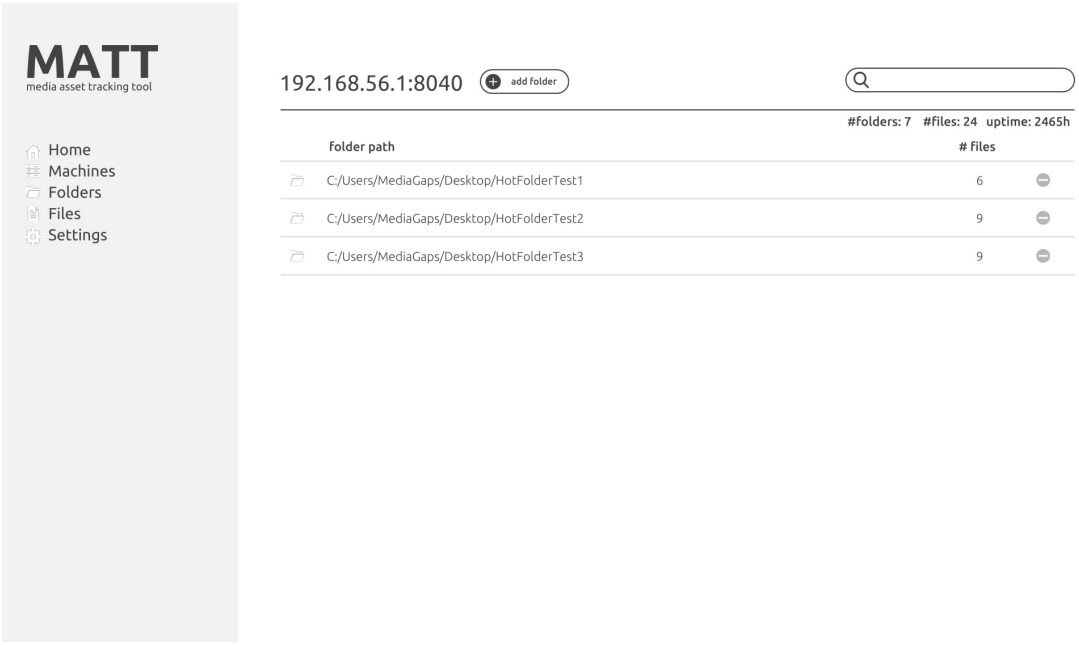

Figure B.3: Machines Details Mockup

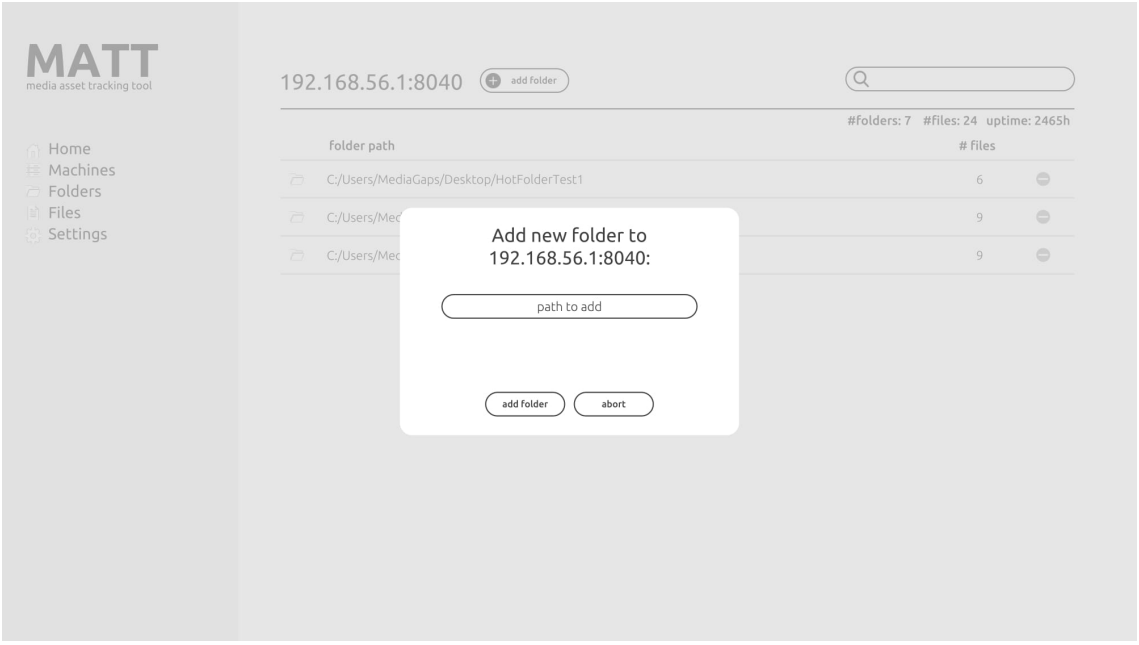

Figure B.4: Folder Addition Details Mockup

B.4 Folders Mockup

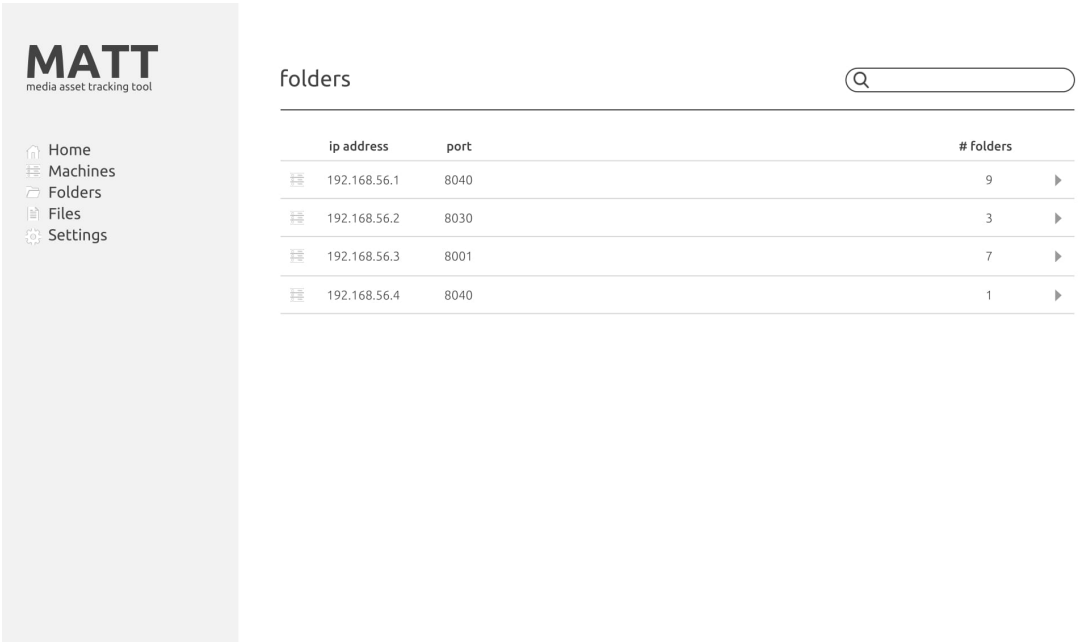

Figure B.5: Folders Mockup

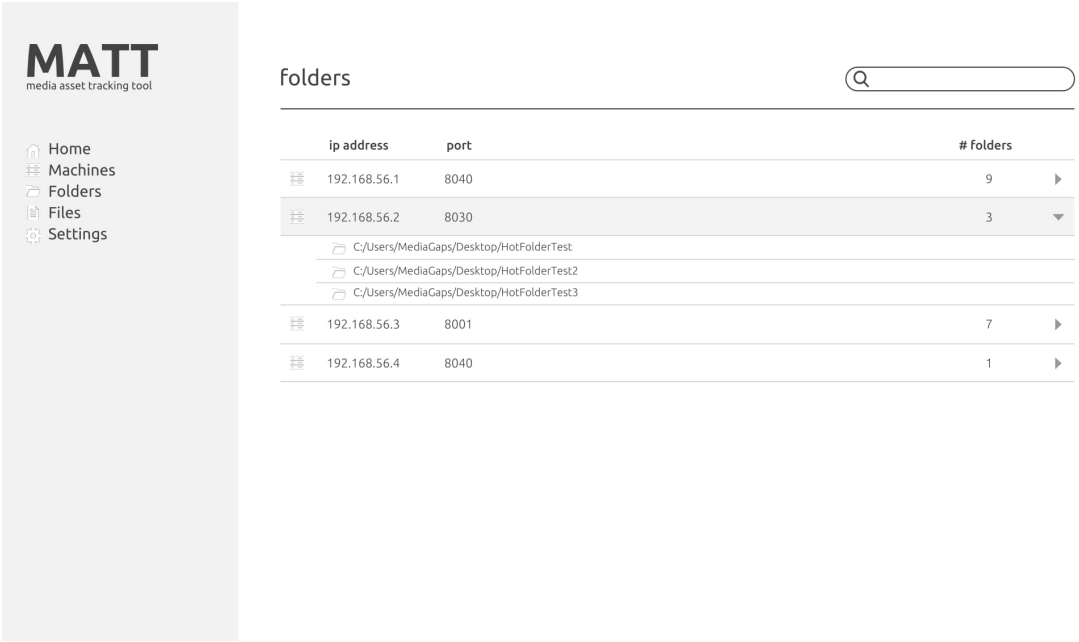

Figure B.6: Folders Expanded Mockup

B.5 Folder Details Mockup

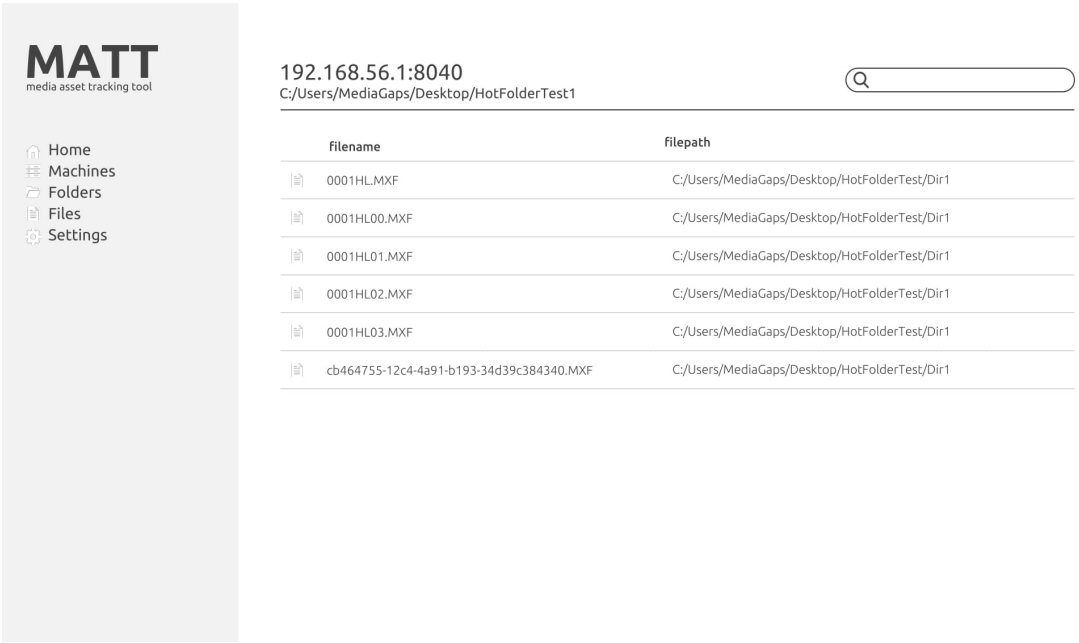

Figure B.7: Folder Details Mockup

B.6 Files Mockup

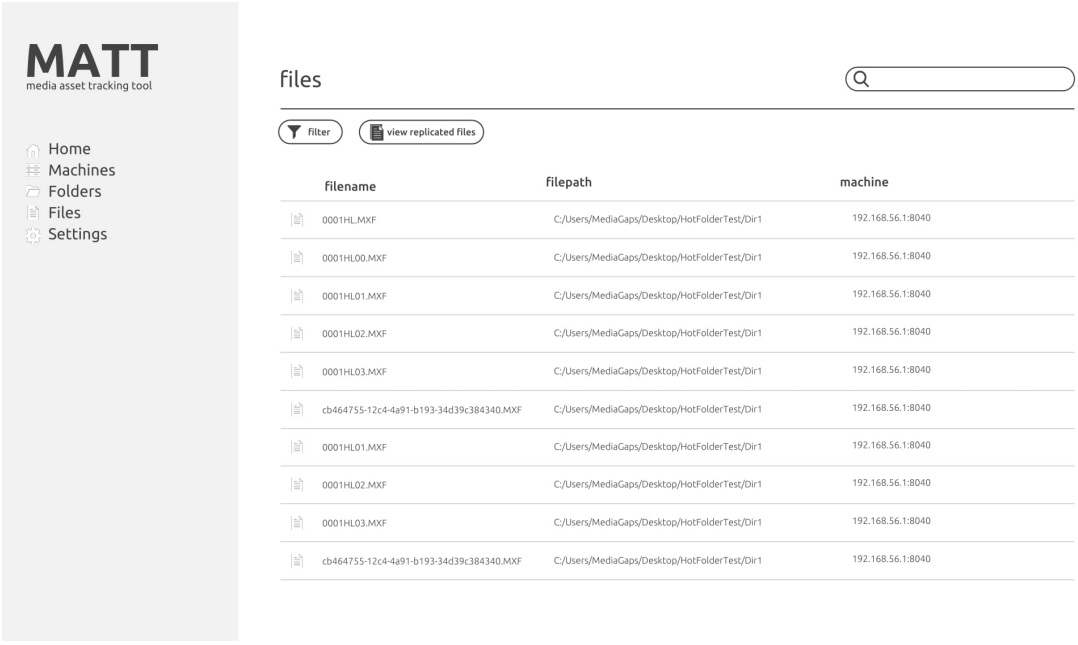

Figure B.8: Files Mockup

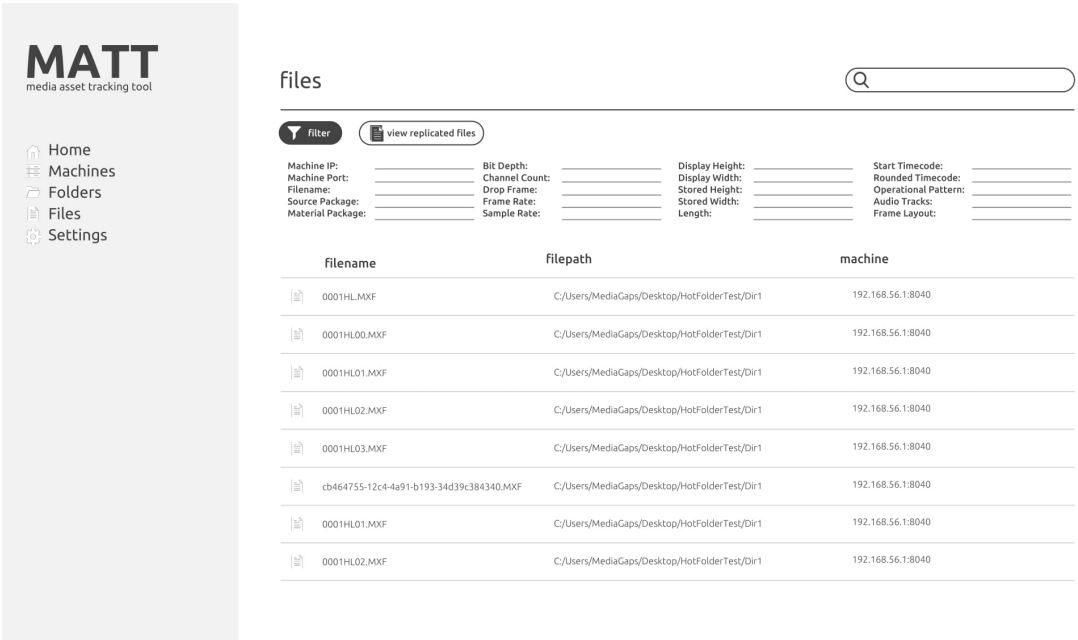

Figure B.9: Files Filter Mockup

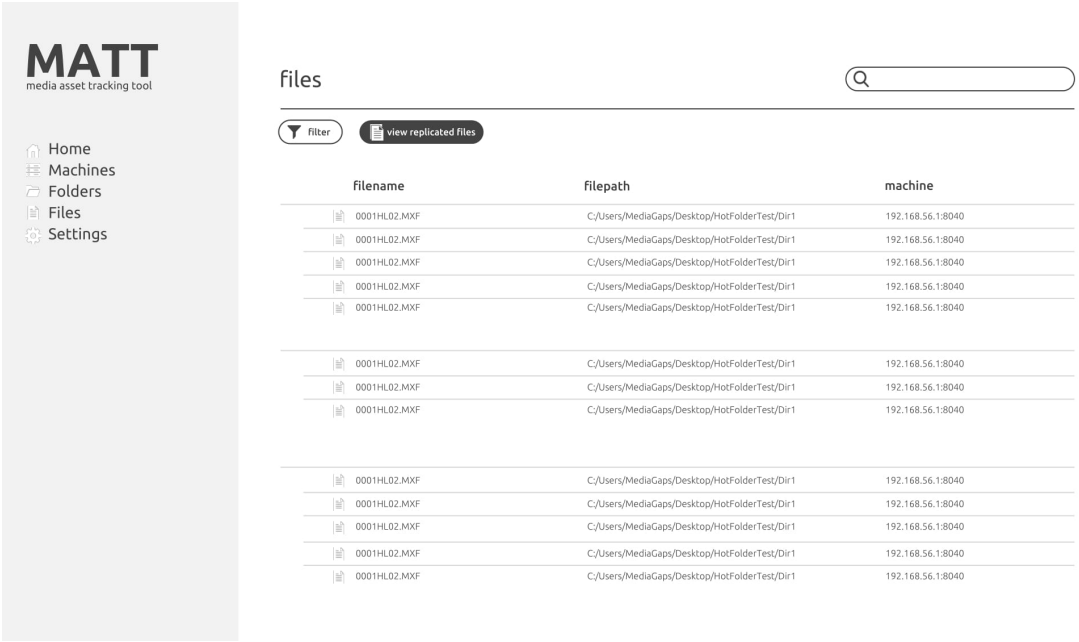

Figure B.10: Files Replicated Mockup

B.7 File Details Mockup

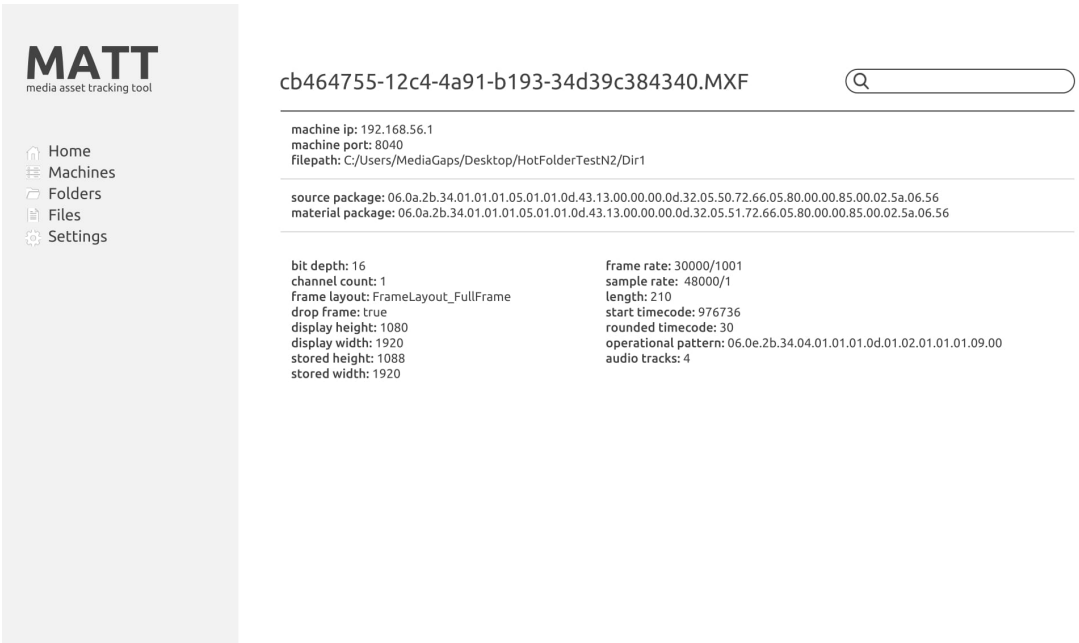

Figure B.11: Files Details Mockup

References

[1] Bruce Devlin, Jim Wilkinson, Matt Beard, e Phil Tudor. *The MXF Book: An Introduction to the Material eXchange Format*. Focal Press, Amsterdam; Boston, 1 edition edição, Março 2006.

[2] Pedro Magalhães. Produção de Televisão UHD para Eventos em Direto. Tese de mestrado, Universidade do Porto, Julho 2016.

[3] David Austerberry. *Digital Asset Management*. Taylor & Francis, Julho 2012.

[4] Chris Richardson. Building Microservices: Using an API Gateway, Junho 2015. URL: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/.

[5] Sebastián Peyrott. An Introduction to Microservices, Part 3: The Service Registry, Outubro 2015. URL: https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/.

[6] Kong - Open Source API, 2017. URL: https://getkong.org/.

[7] Miguel Poeira. *Virtualização de Estúdios Móveis na Produção de Conteúdos Audiovisuais em Direto*. Tese de doutoramento, Universidade do Porto, Junho 2016.

[8] ST 377-1:2011 - SMPTE Standard - Material Exchange Format (MXF) #x2014; File Format Specification. *SMPTE ST 377-1:2011*, páginas 1–183, Junho 2011. doi:10.5594/SMPTE.ST377-1.2011.

[9] Deepak Mohan. *The evolving value chain in the television industry : changes in pay TV delivery and its implications for the future*. Thesis, Massachusetts Institute of Technology, 2014. URL: http://dspace.mit.edu/handle/1721.1/90718.

[10] Ricardo Serra. Interfaces tácteis baseadas em HTML5/CSS3/JavaScript. Tese de mestrado, Universidade do Porto, Julho 2011.

[11] Pedro Ferreira. MXF- a progress report. 2010. URL: https://tech.ebu.ch/docs/techreview/trev_2010-Q3_MXF-1.pdf.

[12] M. Yousif. Microservices. *IEEE Cloud Computing*, 3(5):4–5, Setembro 2016. doi:10.1109/MCC.2016.101.

[13] D. S. Linthicum. Practical Use of Microservices in Moving Workloads to the Cloud. *IEEE Cloud Computing*, 3(5):6–9, Setembro 2016. doi:10.1109/MCC.2016.114.

[14] Chris Richardson. Pattern: API Gateway / Backend for Front-End, 2017. URL: http://microservices.io/patterns/apigateway.html.

[15] Qusay Hassan. Demystifying Cloud Computing. *CrossTalking*, Janeiro 2011. URL: http://static1.1.sqspcdn.com/static/f/702523/10181434/1294788395300/201101-Hassan.pdf?token=74D21PKHLEH4MrgEBhPA4oTJpbA%3D.

[16] Peter Mell e Timothy Grance. The NIST Definition of Cloud Computing. *NIST Special Publication 800-145*, Setembro 2011. URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[17] L. Vokorokos, M. Uchnár, e L. Leščišin. Performance optimization of applications based on non-relational databases. Em *2016 International Conference on Emerging eLearning Technologies and Applications (ICETA)*, páginas 371–376, Novembro 2016. doi:10.1109/ICETA.2016.7802079.

[18] Carlos Soares. NoSQL (NoRDBMS) - Slides da UC Sistemas de Informação e Bases de Dados, 2016.

[19] Apache Software Foundation. Apache Cassandra, 2017. URL: http://cassandra.apache.org/.

[20] Datastax. Datastax Apache Cassandra, 2017. URL: http://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling.

[21] CPlusPlus. cplusplus.com. URL: http://www.cplusplus.com/info/.

[22] ISO C++. ISO C++. URL: https://isocpp.org/std/status.

[23] André Restivo. HTML5, 2017. URL: https://web.fe.up.pt/~arestivo/presentation/html5/#1.

[24] W3C. HTML & CSS - W3c, 2017. URL: https://www.w3.org/standards/webdesign/htmlcss.

[25] MDN. HTML5 - Web Developer guides | MDN, 2017. URL: https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5.

[26] André Restivo. CSS 3, 2017. URL: https://web.fe.up.pt/~arestivo/presentation/css3/#1.

[27] MDN. CSS | MDN, 2017. URL: https://developer.mozilla.org/en-US/docs/Web/CSS.

[28] MDN. JavaScript | MDN, 2017. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[29] André Restivo. Javascript, 2017. URL: https://web.fe.up.pt/~arestivo/presentation/javascript/#1.

[30] Node.JS. Node.js, 2017. URL: https://nodejs.org/en/about/.

[31] React. A JavaScript library for building user interfaces - React, 2017. URL: https://facebook.github.io/react/.